THESIS

DISTRIBUTED EMULATION IN SUPPORT OF LARGE NETWORKS

by

Brian Greunke

June 2016

Thesis Advisor: Robert Beverly
Second Reader: Justin P. Rohrer

REPORT DOCUMENTATION PAGE

Form Approved OMB No. 0704–0188

1. AGENCY USE ONLY *(Leave Blank)*	2. REPORT DATE June 2016	3. REPORT TYPE AND DATES COVERED Master's Thesis 07-05-2015 to 06-01-2016

4. TITLE AND SUBTITLE DISTRIBUTED EMULATION IN SUPPORT OF LARGE NETWORKS	5. FUNDING NUMBERS N66001-2250-5823
6. AUTHOR(S) Brian Greunke	
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Naval Postgraduate School Monterey, CA 93943	8. PERFORMING ORGANIZATION REPORT NUMBER
9. SPONSORING / MONITORING AGENCY NAME(S) AND ADDRESS(ES) Department of Homeland Security 245 Murray Lane SW, Washington, DC 20528	10. SPONSORING / MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

The views expressed in this document are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government. IRB Protocol Number: N/A.

12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited	12b. DISTRIBUTION CODE

13. ABSTRACT *(maximum 200 words)*

Network emulation is a valuable, though potentially resource intensive, method for virtualizing networks for analysis or testing. Though high-powered servers are becoming increasingly accessible, the size and complexity of physical networks have increased in a similar fashion, thereby limiting the type and size of networks that can be emulated on a single physical machine. In this thesis, we present a tool that allows the developers of ground truth topologies to distribute the emulation requirements across multiple physical machines, thereby increasing the size of networks that can be emulated. First, we reexamine existing tools to discover current methods for emulating synthetic and physical networks. Then we modify an existing platform to enable execution on multiple machines, while increasing flexibility for future extensions. We then develop methods for efficiently distributing the topology among the available resources in order to maximize the potential scale. Finally, we run a series of scenarios simulating real world events, such as a Border Gateway Protocol (BGP) hijack attack, in order to demonstrate the utility and efficiency of the system.

14. SUBJECT TERMS distributed emulation, network emulation, linear program, BGP hijack	15. NUMBER OF PAGES 75
	16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UU

NSN 7540-01-280-5500

Standard Form 298 (Rev. 2–89)
Prescribed by ANSI Std. 239–18

THIS PAGE INTENTIONALLY LEFT BLANK

DISTRIBUTED EMULATION IN SUPPORT OF LARGE NETWORKS

Brian Greunke
Captain, United States Marine Corps
B.S., University of Wisconsin-Green Bay, 2007

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN COMPUTER SCIENCE

from the

NAVAL POSTGRADUATE SCHOOL
June 2016

Approved by: Robert Beverly
Thesis Advisor

Justin P. Rohrer
Second Reader

Peter Denning
Chair, Department of Computer Science

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

Network emulation is a valuable, though potentially resource intensive, method for virtualizing networks for analysis or testing. Though high-powered servers are becoming increasingly accessible, the size and complexity of physical networks have increased in a similar fashion, thereby limiting the type and size of networks that can be emulated on a single physical machine. In this thesis, we present a tool that allows the developers of ground truth topologies to distribute the emulation requirements across multiple physical machines, thereby increasing the size of networks that can be emulated. First, we reexamine existing tools to discover current methods for emulating synthetic and physical networks. Then we modify an existing platform to enable execution on multiple machines, while increasing flexibility for future extensions. We then develop methods for efficiently distributing the topology among the available resources in order to maximize the potential scale. Finally, we run a series of scenarios simulating real world events, such as a Border Gateway Protocol (BGP) hijack attack, in order to demonstrate the utility and efficiency of the system.

THIS PAGE INTENTIONALLY LEFT BLANK

Table of Contents

THIS PAGE INTENTIONALLY LEFT BLANK

List of Figures

THIS PAGE INTENTIONALLY LEFT BLANK

List of Acronyms and Abbreviations

ACL	Access control list
AS	Autonomous System
ASN	Autonomous System Number
API	Application Programming Interface
BA	Barabási-Albert
BGP	Border Gateway Protocol
CAIDA	Center for Applied Internet Data Analysis
CoIN-OR	COmputational INfrastructure for Operations Research
CPU	Central Processing Unit
DERIK	Distributed Emulation Router Inference Kit
DoD	Department of Defense
eBGP	External Border Gateway Protocol
ERIK	Emulated Router Inference Kit
GB	Gigabyte
GHz	Gigahertz
GLPK	GNU Linear Programming Kit
GNS3	Graphical Network Simulator - 3
HPC	High Performance Computing
iBGP	Internal Border Gateway Protocol
ICMP	Internet Control Message Protocol

IO	Input-Output
IOS	Internetwork Operating System
IP	Internet Protocol
ISP	Internet Service Provider
LTE	Long Term Evolution
MB	Megabyte
MIPS	Microprocessor without Interlocked Pipeline Stages
MRT	Multi-Threaded Routing Toolkit
NPS	Naval Postgraduate School
NPE	Network Processing Engine
OPNET	Optimized Network Engineering Tool
OS	Operating System
PoP	Point-of-Presence
RDECOM	Research, Development and Engineering Command
RFC	Request for Comments
SSH	Secure Shell
UDP	User Datagram Protocol
UML	Unified Modeling Language
U.S.	United States
VDE	Virtual Distributed Ethernet
VPN	Virtual Private Network
VM	Virtual Machine

Acknowledgments

I'm profoundly grateful for my wife, Stephanie, for her understanding and support thoughout my time at NPS. Without her sacrifices during the past two years, I could never have accomplished this. I also need to thank my parents for imparting me with a lifelong desire to learn and the opportunity to do so.

My advisor, Dr. Robert Beverly, provided an incredible amount of guidance, patience, and expertise and I feel honored to have learned from him. His knowledge and passion are obvious, and provide an example for all to emulate. Dr. Justin P. Rohrer was an invaluable second reader, knowledge base, and source of many insights. Thank you to Erik Rye for the advice and assistance, before, during, and after NPS.

I'd also like to thank Dr. Ralucca Gera and Dr. Robert Dell. Their instruction both inspired my work and contributed to the quality of the end product.

Finally, I'd like to thank my classmates for making the past two years incredibly enjoyable.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 1: Introduction

Virtual networks, both emulated and simulated, are important tools for network research. For example, a network researcher may be interested in analyzing how a worm propagates through a network, or a network administrator may want to view policy change impacts prior to deployment. In both instances, best practices dictate that a non-production network is required to facilitate such exploration and understanding without impacting the production network. Additionally, in order to have reproducible results, a known network, which can be easily and identically recreated, is necessary. Often, a large network is required to fully analyze a problem, however creating a physical test bed of non-trivial size is frequently time and cost prohibitive. To circumvent these constraints, researchers or administrators can turn to network simulation or network emulation.

As the size and scope of real world networks increases, the resources required to efficiently execute emulations and simulations of these networks also increases. Though available computing power of commodity hardware continues to grow, the scale and complexity of networks is also increasing at a commensurate rate. Clearly, we do not have access to unlimited resources.

We term a collection of routers and their interconnection a network "topology." Many problems, such as the analysis of Border Gateway Protocol (BGP) behavior, a subject we examine as a case study in this thesis, can benefit from large interconnected topologies. The real-world complexity of a protocol such as BGP may be better analyzed using an emulated topology versus a simulated topology. The development of emulation architectures that can scale to the size of the network topology and efficiently use the resources available is a challenging task and is the focus of this thesis.

Administrators aim to simplify development, expedite execution, and mitigate risk in the testing and analysis of their networks. Virtualized networks are often for these tasks [1]. Within a controlled environment, modifications to a network, protocol, or model can be executed – and the effects measured – without affecting real-world users or services. In a similar vein, researchers also require large networks with which they can analyze, modify, or develop against [1]. Developing a ground-truth network topology to determine expected behaviors reduces the complexity of analysis and reduces the potential for unexpected side effects. Additionally, using virtual networks allows a large space of scenarios, e.g., traffic models or faults, to be executed in an automated fashion, thereby improving the thoroughness of the analysis. Automating the model creation, network creation, and analysis can greatly improve the efficiency of scenario development and enables common practices like Monte Carlo analysis [2]. For example, Costatino et al. used both network simulation and Monte Carlo analysis to produce their results when analyzing performance of Long Term Evolution (LTE) gateways [3].

Many research scenarios allow problems to be represented mathematically using formally defined models. For example, one could represent a graph using a simple adjacency matrix and execute any number of previously defined algorithms against said matrix. In these instances network simulation can provide an efficient method for executing these mathematical models, and can be an efficient way to test and analyze newly developed scenarios. Complex behaviors can be programmatically executed, and efficiently simulated – often using a single physical computer.

While simulation is a valuable tool, further specificity may be desired and simulators may not capture the true, nuanced behavior of actual and deployed systems. For example, administrators of large networks may enforce routing policies to maximize profits. Such policies may contradict expected protocol behaviors. A BGP simulator which implements behavior based only on the defined standard may always choose the path with the fewest Autonomous System (AS) hops, while an Internet Service Provider (ISP) may prefer a path with the most monetary gain. The Request for Comments (RFC) for BGP contains “suggested” and “may” behaviors, such as support for or values of BGP timers. Without visibility into a proprietary implementation, there is no way to know how the protocol is actually implemented. These values will affect network behavior such as packet propagation and convergence time, and are difficult to capture using a simulator. Network emulation is a technique for simulating network properties by mimicking specific component behavior [4]. Network emulation can provide real-world behavior for network scenarios and more closely copies the behavior of actual, physical network devices. Often, proprietary implementations of a protocol are a “black-box” and simulating these implementations is challenging. Emulation uses the actual behavior of the proprietary software and removes the attempt at *copying* implementation specifics. Additionally, versus simulation, emulation has several added benefits such as: allowing actual network traffic to traverse between the virtual network and real world networks; and deploying the actual behavior of any operating system software.

Many emulation and simulation tools exist. Though, as the size and complexity of real world networks grow, the physical limitations of these tools may be reached. Emulation often requires resources equal to the physical hardware being emulated, plus some amount of overhead. This resource utilization limits the number of instances which can be emulated on a given physical architecture and requires careful consideration when creating emulated networks. Though thoughtful software implementations can mitigate the amount of resources required, many software packages are developed to run on a single physical instance, and are unable to span multiple physical machines. These limitations coupled together, constrain the scale to which we can emulate network topologies.

1.1 Motivation

The primary motivation within this thesis is to enable existing network emulators to run at an order of magnitude larger scale than currently possible. While much can be gained by conducting research on small, representative networks, in many cases constructing a derived network similarly sized to the desired network is preferable or necessary. With a surplus of computing resources available in the form of powerful servers or Infrastructure as a Service, we strive to remove the constraints imposed by the resources available on a single, physical machine or by software coupled to a single machine, providing researchers a valuable option for virtualizing large networks.

1.2 Research Questions and Contributions

In our work, we investigate these primary questions:

1. How can we emulate networks an order of magnitude larger in size than currently possible?
2. How can we best automate virtual network creation, and efficiently distribute emulated network elements across heterogeneous physical resources?

3. What is the extent to which more complex and realistic Internet-like topologies can be modeled on the emulated network?
4. As a concrete application to run on the emulated network, can we model a BGP hijack attack?

We contribute to and extend the state-of-the-art though the following:

1. We develop DERIK, an automated tool that can generate a large, virtualized network based on an existing graph model, distribute the emulated topology across a number of physical servers, and extend the ability to generate analysis scenarios.
2. We develop a linear program that optimizes the distribution of the emulated devices across physical machines to maximize the scale of the networks able to be emulated
3. We demonstrate up to a 37% reduction in traffic on the physical link compared to a uniform distribution in our experiments.
4. We develop a proof-of-concept BGP hijacking application, which demonstrates the utility and effectiveness of Distributed Emulation Router Inference Kit (DERIK).

1.3 Thesis Structure

This remainder of this thesis is structured as follows:

1. In Chapter 1, we discuss the application of distributed emulation, its challenges, and our motivations.
2. In Chapter 2, we investigate existing network virtualization and distribution techniques.
3. In Chapter 3, we describe an upgrade to an existing tool we created to create large scale, distributed emulated network topologies.
4. In Chapter 4, we discuss the results we obtained from analyzing the efficiency of our distributions, as well as the scale to which we were able to build our emulated topologies.
5. In Chapter 5, we discuss our conclusions and possibilities for future work.

CHAPTER 2:
Background and Related Work

This thesis significantly modifies and extends an existing system to enable distributed emulation of large-scale network topologies. We first discuss simulation, emulation, and methods of network virtualization. Next, we discuss the choice of systems to extend and our motivation for selecting a particular emulation platform. We then discuss methods used previously for distributing computing requirements in other software and technical systems. Finally, as our system uses linear programming to optimize the distribution of resources among physical assets, we briefly review optimization problems, and ways in which these methods have been applied to previous software platforms.

2.1 Existing Network Virtualization Tools

Many tools have been developed in an attempt to reduce setup time, reduce execution time, or reduction configuration required for the virtualization of networks. These tools can be generally segmented into two classes: simulation software and emulation software, each with its own subsets of capabilities and limitations. Both classes contain robust tools which can provide high quality virtualizations with regards to the specific desired characteristics of a chosen problem and can provide value to a network administrator, researcher, instructor, or student. The selection of one over another is dependent on the resources available and problem being analyzed.

2.1.1 Simulation tools

Simulation software can be an efficient method for inferring patterns in network models. Multiple layers of the network stack such as: environmental characteristics, mobility patterns, traffic patterns, or routing protocols can be pragmatically modeled using simulation software [5]–[7]. Analyzing models and patterns in virtual networks can be performed at several levels of abstraction using simulation methods. In the simplest form of simulation, mathematical or analytical models can be created and analyzed using software such as MatLab [8]. As an example, networks can be represented as a weighted adjacency matrix, with edges represented by matrix values greater than zero. A desired algorithm, such as the Bellman-Ford algorithm [9], can be executed on the matrix and the results obtained relatively easily. Though these methods can be simply implemented, many real-world network considerations, such as routing policies and implementation details, may be removed by such an abstracted representation. For example, without added configuration and complexity, the effects of packet loss due to interference or congestion will be missing from a simulated environment. Adding this nuanced behavior into a mathematical model is possible, but can become cumbersome or overly complex as additional layers of a stack are included.

As an extension to mathematical modeling, specially designed network simulators can provide granularity to scenarios where purely analytical methods may overly simplify; or can be used to validate analytical models [10]. These tools have pre-developed layers and models which can be used by a developer to more fully flesh the behaviors of the protocol or platform being simulated. Tools such as ns-3 have been developed to provide a higher granularity in network simulation scenarios. ns-3 is a discrete-event network simulator and provides a simulation environment where researchers can use pre-defined models in combination with desired parameters to analyze network environments and scenarios [11]. A developer can simulate and customize multiple layers of the network stack to provide a robust simulation environment Additionally, much of the platform is extensible, allowing users to thoroughly customize the scenarios developed [11]. Feng et al. developed an extension to the ns-3 family of software, ns-BGP, which was created in order to enable ns-2 to simulate BGP routing [12]. Multiple other efforts have been made at simulating inter-domain routing including BSIM [13], BGPSIM [14], and an algorithmic approach detailed by Gill et al. [5]. Optimized Network Engineering Tool (OPNET) is another robust simulation tool and has been used to model a variety of layers and protocols including queue management [15] or comparison of internal routing protocols [16].

Some network simulators are scalable, and are able to simulate large network topologies, though they are often limited by some combination of time and available resources. For example, ns-3 has been evaluated for single-threaded performance and, though efficient as compared to other simulators, is still constrained by the available memory in a system [17]. In Weingartner et al.'s scenario [17], the researchers were able to simulate a 3000 node network which utilized approximately 60MB of memory and their results indicate a linear relationship between the number of nodes and memory required which places an upper limit on the network size with regards to the available memory in a system.

In order to increase scalability and remove the constraints of a single physical machine, advances to simulation systems have been developed to utilize parallelization methods. Research has been conducted on using message passing methods, specialized hardware, and customized routing protocols in conjunction with ns-3 in order to increase the scale and execution speed of simulated network topologies [18]–[21]. Using High Performance Computing (HPC) platforms researchers created simulated networks of $36,044,800$ nodes [20], $750,000$ nodes [19], and 10^9 nodes [21] respectively. These results demonstrate a significant increase in the scale of simulated networks by distributing resources across multiple physical hosts.

2.1.2 Limitations of simulation software

Simulation is often adequate for simulating simple characteristics of networks or protocols but may fall short in simulating a full implementation of a complex system. Inter-domain routing protocols are an example where simulation can be especially challenging. The BGP RFC itself is over 100 pages long. Accurately simulating all the prescribed behaviors defined within the BGP is a demanding task. Per the RFC, parts of the implementation, such as HoldTime [22], of the BGP standard are suggested rather than required. Therefore, we are unable to truly simulate a given proprietary "Black-box" vendor implementations of said standard. Additionally, much of true BGP behavior in real-world networks is the result of implementation specific differences, which adds further complexity to the simulation.

Though simulation can be a robust method for virtualizing networks, simulation is truly an estimate, and can rarely capture the true behavior of real world networks [23]. Only the behaviors specifically defined within the simulation are executed, possibly removing undefined behaviors or second order effects. Simulating proprietary platforms presents further challenges. These systems may be a *Black Box*, where the developers of the simulation lack access to the actual implementations and can only observe the input and output. Rampfl et al. describe several instances of simulations providing high quality representations of reality where there are also notable differences in certain measurements and note the requirement for credibility and validation of simulation models [23]. The trust one can place on these simulations is then a function of the validation and testing done on a model or simulation coupled with the amount of modifications made to the model for an individual instance. Any extension of the simulation or model also relies on the correctness of the underlying model [23].

Though a simulation may be accurate for a particular set of features or functionality, it may lack the complete feature set of a real world implementation. Packet tracer is a popular simulation tool often used in academic environments. It provides a simple, graphical interface for creating, interacting with, and analyzing networks and network traffic. Though simple to use, it lacks simulated features and behaviors implemented in real-world Cisco platforms such as Internal Border Gateway Protocol (iBGP) [24].

Simulation software also faces scalability limits as discussed earlier in section 2.1.1. Representation of nodes and traffic requires memory resources and both the execution and analysis of a simulated topology requires processor time. As the size of a topology grows, the memory required, time to execute, and processing requirements can increase. Therefore this constrains an administrator to either: decrease the size of the network to fit within the available physical resources, provide more physical resources to accommodate the size of the network, or potentially increase the execution time.

2.1.3 Emulation software

Emulating network devices is a useful method for obtaining near real world behavior without the complications involved with the procurement, installation, and management of physical hardware. Emulation software works by providing an abstraction layer between physical hardware and some software which expects to run on a different hardware platform. This allows the execution of said software on hardware not originally intended for this use. This is analogous to commonly used "Virtual Machines", or virtual computers running real operating systems via hypervisors such as Virtualbox [25]. Additionally, emulation software provides the capability of interacting with actual network traffic by injecting real network traffic into the emulation environment, exporting generated traffic into a real network, or both.

The software package Quagga [26] is popular choice for emulating routing on Unix systems. Quagga provides an abstraction layer between Unix hardware and Quagga clients. It supports many popular routing protocols and is extensible for custom administration [26].

Graphical Network Simulator - 3 (GNS3) is a software suite which provides a graphical user interface for creating emulated networks [27]. In order to emulate Cisco devices, GNS3 uses Dynamips, a tool created to emulate the Microprocessor without Interlocked Pipeline Stages (MIPS) instruction set [28]. Dynamips runs actual Cisco Internetwork Operating System (IOS) binaries which operates identically to the IOS running on physical hardware. None of the behavior controlled by software is abstracted or removed by an emulated router as opposed to a simulated instance. Dynamips provides an Application Programming Interface (API), which is used by the user interface of GNS-3 to create emulated Cisco devices in the background. While this is a powerful and useful tool for easily creating emulated networks, it is not designed for the automation of network creation and creates some constraints on the size of topologies which can be created. GNS3 uses a graphical user interface for the creation of a network and much of the work is done by clicking, dragging, and then manually configuring an emulated instance. This can be beneficial for small, simple networks and for less experienced administrators, but becomes tedious or impractical when trying to create many networks of many nodes.

Autonetkit is another emulation platform that attempts to remove some of the limitations of tools like GNS3 [29]. AutoNetkit deploys an emulated network automatically, without required configurations, based on a provided model. It can automatically create required configuration files and build a fully functioning emulated network, with limited effort required by an administrator. AutoNetkit uses User-Mode Linux kernel and allows models to be inputted in several formats, such as GraphML, reducing the requirement for a specific user interface [29]. In order to scale the size of the emulated topologies, AutoNetkit uses Virtual Distributed Ethernet (VDE) to create virtual connections between physical devices and enable the deployment of a topology across multiple, physical boxes [29]. Additionally, reference [30] indicates that AutoNetKit can deploy to Dynamips, but upon testing the software, the authors were unable to evaluate this described functionality due to a lack of documentation. We downloaded, installed, and executed AutoNetKit for a given topology and analyzed the results but were unable to find the required scripts for deploying to Dynamips. The authors indicate that the software allows the use of multiple servers to run the topology but we were unable to locate publications on the methods, scale, or performance of such.

Emulated Router Inference Kit (ERIK) was designed as a platform to create ground truth emulated networks, which could be used to automate topology inference [28]. ERIK is written in Python and, similar to AutoNetkit, creates emulated topologies using a pre-created model as input [28]. Additionally, ERIK automates the processes of network modification and network measurement by leveraging additional capabilities of Dynamips and the inference software scamper [31]. ERIK takes in a model, creates the required Dynamips commands, executes the commands, and then modifies the network based on a set of pre-defined scenario scripts. Throughout the process, multiple vantage points (in the form of a virtual host running on the same server that changes virtual network attachment point) are used to measure the results of the network modifications. ERIK serves as the foundation for this thesis.

2.1.4 Limitations of current emulation tools

Emulation of network devices can be a resource intensive endeavor. Often, the emulated device requires the full amount of memory normally available by the physical hardware which it expects, to be allocated continually, creating a hard limit on the number of emulations a physical device is capable of. For example a Cisco c7200 series router with an Network Processing Engine (NPE) NPE-400 has a default of 128Megabyte (MB) and is expandable up to 512MB for the NPE alone. The amount of memory used on a platform at any given time is a function of the IOS running, the number of type of network modules installed, and the amount of traffic processed. If the memory requirement of a router is not realized, they often do not fail gracefully, therefore creating a need for consideration in deployment. Routing can also be a processor intensive operation. Without careful allocation by the hypervisor, emulated devices can use continuous high levels of processing power from their physical host, quickly using the entire capacity of the physical host. Dynamips requires the manual configuration of an idle-pc value, which limits the amount of host processor used by the emulated device.

At the time of writing, some existing emulation software expects the entirety of a network topology to be located on a single physical machine. Stephen Guppy, the CEO of GNS-3, indicates that a “*multi-tenancy*” feature is currently in development, but was not available at the time of writing [32]. ERIK creates execution scripts which are expected to be run on a single host and does not provide the interconnections to deploy on multiple hosts. The requirement to emulate the entire topology on a single host limits the number of emulated instances which can be created. For example, Rye et al. used a physical server with 16, 1.2Gigahertz (GHz) cores and nearly 99Gigabyte (GB) of memory; and was able to create a topology of 300 emulated Cisco routers before running into memory constraints [28].

Even if software allows a topology to be distributed across multiple hosts, it may not be entirely clear how to optimally distribute said topology. For example, minimizing the number of virtual links which traverse physical connections may be required, or minimizing the amount of traffic which traverse these links may be required. As the size and scale of the emulated network grows, these considerations will become more important to ensure that a topology can be emulated on a set of physical hosts without surpassing any of the resource capabilities of said hosts.

2.2 Distribution Methods

The ability to distribute resource requirements across a cluster of resources has been applied in many fields of computer science. Parallel computing, concurrent computing, and distributed computing all have methodologies which can be applied toward scaling network virtualization.

Fujimoto et al. [10] discuss two methods for scaling simulations: time parallel simulation and parallel discrete events. While time parallel simulation can decrease runtime, it does not allow the size of the network simulated to increase. In contrast, the parallel discrete events method, as described by the authors, allocates parts of a simulated topology to a processor. Thus, parallelizing discrete events is a preferred method for scaling the size of topologies as it allows both an increase in size, and a decrease in execution time [10]. Fujimoto et al. further discuss the use of multiple instances of a simulator working on disparate parts of a simulation, which they deem a “federated simulation” [10] and discuss the benefits of such. In these specific instances the authors need to account for specific simulation events, which will not be present in emulated environments as the software running on the emulation platform will manage these events as designed.

Yocum et al. discuss “topology partitioning” [33], a method to distribute parts of a network topology across multiple physical resources. The methodology used by Yocum relies on techniques from graph theory graph partitioning including k-cluster [34] and METIS [35] to distribute a topology across physical resources. The authors used the ModelNet [36] emulation platform to analyze their partitioning methods and their analysis shows that thoughtful partitioning can nearly double the capability versus random assignment [35]. The k-cluster method described returns equally sized connected components where all edges are considered equal and the METIS method returns similarly sized components but is able to use edge weights [33]. These methods are valuable for homogeneous environments, such as distributing load across identical cores of a host, but may not be as applicable for heterogeneous environments containing hosts of varying capabilities. In our methodology described in Chapter 3, we learn from these applications and attempt to apply some of the concepts to a heterogeneous environment.

2.3 Linear Optimization

As defined by Rardin [37], "Optimization models represent problem choices as decision variables and seek values that maximize or minimize objective functions of the decision variables subject to constraints on variable values expressing the limits on possible decision choices." This process can be used to analyze a problem, construct a mathematical model of said problem, and potentially find an optimal solution.

Linear optimization is a specific form of mathematical optimization where the requirements of the model can be represented in a linear relationship [37]. In a feasible linear program, one optimizes the objective function subject to parameters and constraints and achieves a feasible solution as good as any other feasible solution [37]. Such linear programs are expressed using the standard form:

$$\begin{aligned} \text{maximize} \quad & c * x \\ \text{subject to} \quad & Ax \leq b \\ \text{and} \quad & x \geq 0 \end{aligned}$$

where x is a vector of variables, c and b are vectors of coefficients, and A is a matrix of coefficients [37].

Given these inputs, one of several methods, such as the Simplex algorithm [37] can be used to "solve" or find the optimal solution for the problem. Specialized solver software is usually used to solve non-trivial linear programs and is available via open source repositories. The COmputational INfrastructure for Operations Research (CoIN-OR) solver is one such implementation and was used throughout our experiments [38]. Additional free solver packages exists such as GNU Linear Programming Kit (GLPK) [39] and Gurobi [40], though these were not used during our testing.

After a problem is discovered it needs to be properly fleshed out and formulated prior to implementing programmatically. Naval Postgraduate School (NPS) format is a popular method to describe a linear program [41]. In this format, a problem is expressed using the sections: indicies, data, decision variables, and formulation [41]. Indicies are used in a similar fashion to other fields and indicates a specific instance of a variable. Data is problem specific input. Decision variables are the variables which the problem is attempted to solve for. The formulation is the actual problem, expressed as a mathematical model, using the other sections indicated [41]. In Chapter 3 we use this method to describe our problem as it provides a clear written representation of the problem which can then be implemented in whichever programmatic language is desired.

Linear programming is commonly used to analyze problems in routing such as the minimum cost network flow problem. In this specific instance, a linear program determines the minimum cost to traverse from a source to a sink. The network can be expressed as a weighted digraph, where links are represented by arcs and the weight indicates a cost of traverse the arc [37]. This example can be represented by a graph as shown in Figure 2.1. Using a linear program provides a benefit over traditional network path selection algorithms, such as Bellman-Ford [9], as it provides an opportunity to provide additional constraints or input to a problem.

2.4 Challenges in Modeling BGP

BGP is the prevalent protocol for routing traffic between Internet domains and has been widely used for over 20 years. The original BGP RFC was created in 1994 and has since been revised three times, with the latest RFC 4271, containing another six updates [22]. Despite a long and complex standard, due to its widespread use as the inter-domain routing

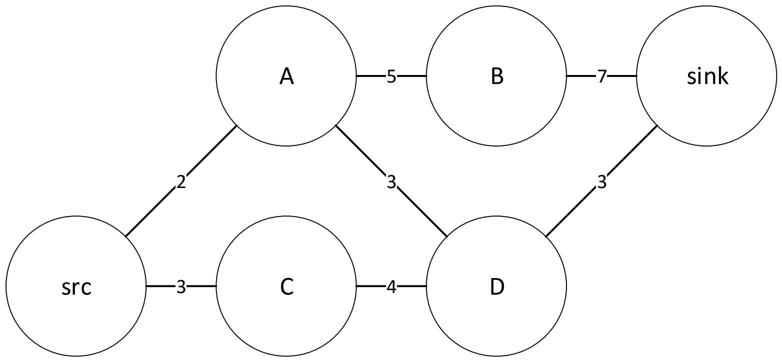

Figure 2.1: A linear program can find an optimal solution for the minimum cost path from the source to the sink. In this case the path would be: source - A - D - sink, for a total cost of 8.

protocol for the Internet, simulating and emulating BGP is both a challenging and useful endeavor.

One aspect which makes BGP simulation and emulation challenging is the common practice of implementation specific policies which can override default behavior. Much of the behavior of the inter-domain traffic within the Internet is derived via business agreements between the organizations who route traffic between their ASs. For example, an organization who is providing Internet service may choose to route traffic over a path with which it receives a higher monetary rate, rather than (following BGP default behavior), a path which may have fewer AS hops.

The sheer size of the Internet is another contributing factor to the challenges of modeling BGP. At the time of writing, Center for Applied Internet Data Analysis (CAIDA) indicates that there are 54130 ASs with 22137 observed relationships [42]. From an emulation perspective this poses two challenges: the platform must have the capability to emulate many devices and it must have the memory and processing capability to support large required routing tables for all devices. Additionally, as Gill et al. discuss, discovering a ground truth is challenging due to both factors mentioned above: the large size of the Internet and proprietary implementations of BGP which are not released publicly [5].

In Chapter 3 we discuss the methods we use to enable an emulated, ground truth network to provide representative behavior for inter-domain routing networks. We build upon existing software platforms and use the principles described in this chapter to build emulated topologies that scale to larger sized networks than previously possible.

THIS PAGE INTENTIONALLY LEFT BLANK

CHAPTER 3: Distributed Emulation Methodology

We recognize the limitations of existing emulation platforms as discussed in Chapter 2, and endeavor to create our own software platform, the Distributed Emulated Router Inference KitDERIK. DERIK is intended to facilitate the creation of large scale emulated network topologies. We aim to:

1. Automate the creation of an emulated topology using a model as input into the system
2. Efficiently distribute the emulated topology across the available physical resources
3. Provide an extensible platform for application development where researchers, teachers, or practitioners can create and analyze test scenarios using the virtualized topology

Toward these ends, we extended an existing tool, Emulated Router Inference Kit ERIK [28]. While ERIK has similar high-level goals and utilizes a similar implementation, it is confined to performing the emulation on a single physical machine. We therefore seek to modify ERIK to enable a single, emulated topology on multiple physical servers. We discuss the specific features provided by ERIK, the current state of DERIK, and the methods we used to efficiently distribute the topology across multiple, physical resources.

3.1 Differences in DERIK from ERIK

- Supports connections between physical servers
- Efficiently allocates emulated devices to physical servers
- Supports templates for scripts and configurations

The work done by Rye in the development of ERIK provided a starting point for automating network topology creation, but still constrained the emulated network to a single physical device. Though emulated devices can communicate between instances of a Dynamips hypervisor, ERIK requires each hypervisor to be located on the same physical host as there is no automatic method for forwarding traffic between physical hosts. Figure 3.1 depicts the structure created by ERIK.

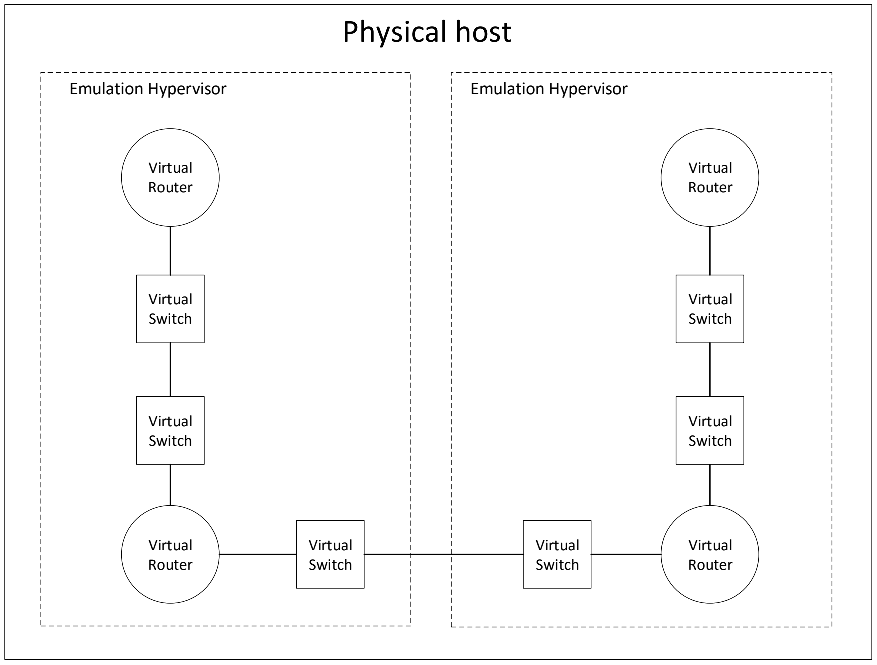

Figure 3.1: ERIK allows interconnections between hypervisors but does not support interconnections between physical hosts.

Verify hypervisor capability using manual processes

Our first task was to ensure that the software ERIK used, specifically Dynamips, was capable of communicating across a physical link. In order to simply verify this, we chose to use ERIK and manual deployment to place a small network on two different physical servers. Then, we would manually configure the physical hosts to forward necessary traffic and create the logical connections required by the emulated devices. This desired system structure is depicted in Figure 3.2.

ERIK requires only the installation of Python2.7 and a Dynamips binary on the physical server where it will be executed. In order to automate the required interconnections between servers, DERIK requires additional packages to be installed on the physical server including: bridge [43], TunTap [44], expect [45], and Tinc [46]. The bridge, TunTap, and Tinc packages are used during the creation of virtual interfaces on the physical server which allow the hypervisor to pass traffic between physical devices. The expect package is used to automate passing commands into the telnet sessions used by Dynamips. Additionally, we use a local

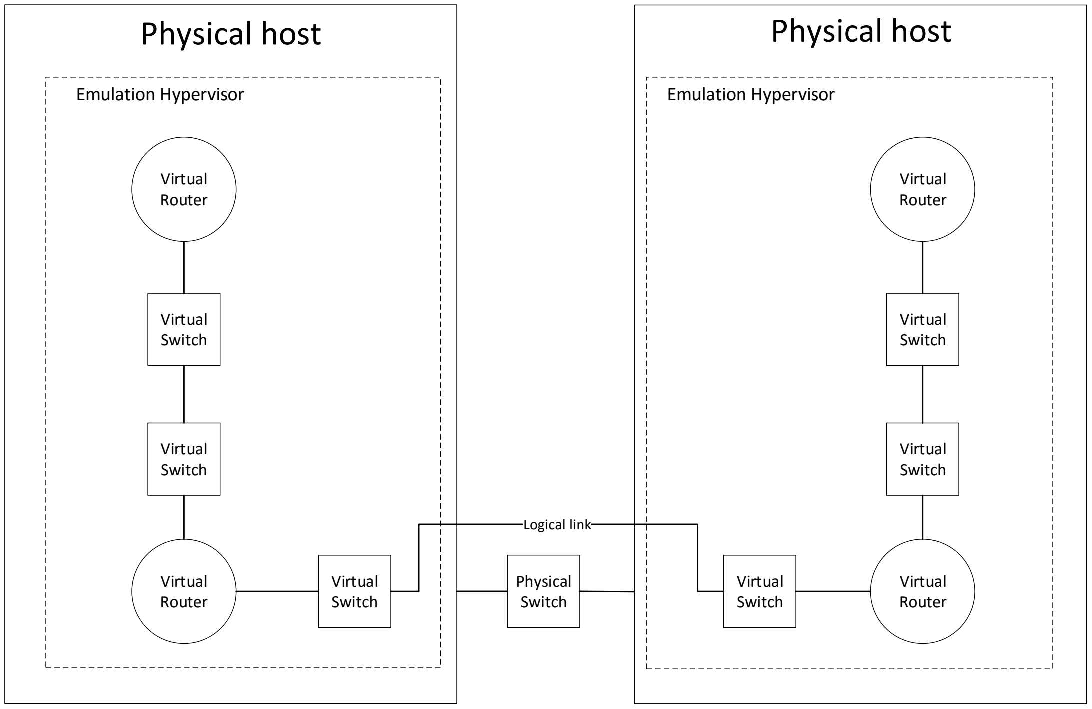

Figure 3.2: A logical link is required between hypervisors which traverses a physical link between hosts and allows the emulated devices to communicate as if they were physically connected.

client (such as a user's laptop) as the build and deployment source for the platform. This local client requires Python2.7 and the following Python packages: Fabric [47], Jinja2 [48], PuLP [49], ipaddress [50], and networkx [51]. Our physical infrastructure consisted of two physical servers running Fedora, and a local client running Windows 10. Server A had 40 cores and approximately 396GB of memory while Server B had 72 cores and approximately 462GB of memory. We built and deployed the topologies from a Microsoft Surface Pro 4 with an Intel Core i7-6650U and 16GB of memory. The servers were connected to one another via a gigabit Ethernet switch and a single network interface on each. The local client was wirelessly connected to the network and accessed each server using Secure Shell (SSH) keys via a management interface on each server.

We used ERIK to create the deployment scripts for a simple, two node network: two routers connected to one another and sharing a BGP adjacency. ERIK created all of the required router configurations and Dynamips commands which we manually transferred and executed on each physical server. Dynamips natively supports the logical connection between an emulated device and its physical hosts using “tap bindings” as a way to bridge traffic between emulated devices and the physical host [52]. ERIK used this feature to connect emulated routers to a virtual host running on the same server. We chose to apply this same methodology for our interconnection between emulated routers on different hosts.

We created a virtual interface called a “tap” interface [44] on each physical host and bound said tap interface to a port on the emulated device via a Dynamips command. We then created a virtual network bridge [53] and Virtual Private Network (VPN) on each physical device using the software suite Tinc [46]. Tinc allows the creation of a VPN between two devices using a set of configuration files on each and a daemon running on each host [46]. By bridging the VPN with the tap interface, we were able to transfer traffic from the tap interface to the VPN, and then across the physical network between the two hosts as depicted in Figure 3.3. Tinc encapsulates the Ethernet frame generated by the router into a User Datagram Protocol (UDP) packet which it then sends across a specific socket. The Tinc daemon on the remote end is monitoring the same socket and decapsulates the packet before forwarding it to a specified interface. The two emulated hosts were able to communicate and unaware that the link between them traversed a VPN.

Multiple inter-server connections

Understanding that any non-trivial network which is distributed between multiple machines would require multiple interconnections, we needed to extend our configuration. In order to create N logical connections which all pass over a single physical link, we were required to create $2N$ virtual interfaces (bridge and tap for each) and N VPNs on each physical host. The creation of multiple tap interfaces was a single command executed on the host but Tinc required: separate configuration directories for each VPN, distinct ports for each VPN, and multiple instances of the daemon in order to differentiate traffic destined for a specific VPN as demonstrated in Figure 3.4

We deemed the use of encryption for the VPNs unnecessary as our physical network was secured behind a firewall and the traffic within our emulated network was limited to test data. Additionally, encryption would hinder our efforts at analyzing network traffic which traverses the physical interfaces. Therefore, we removed the cipher and digest selections

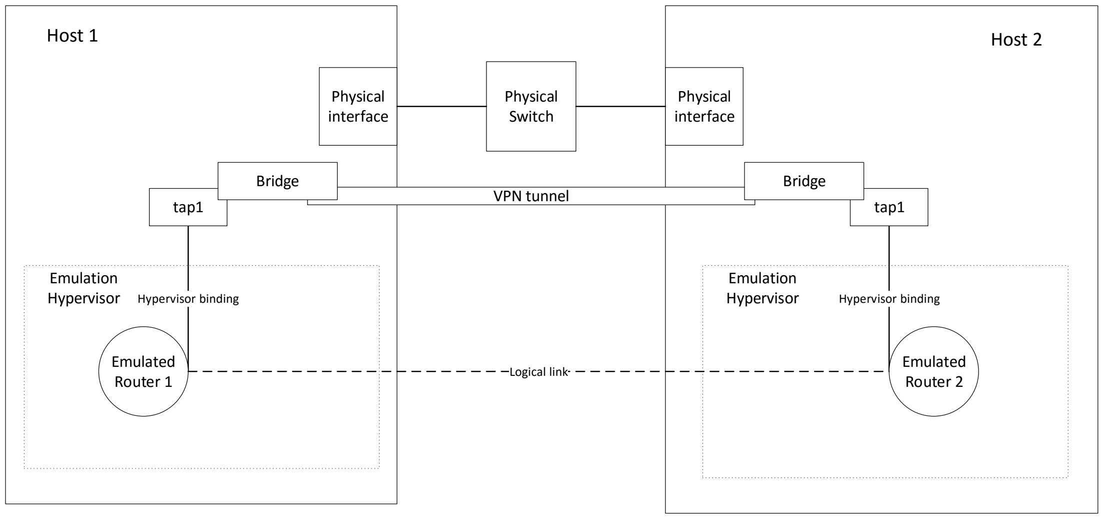

Figure 3.3: We create a tap, bridge, and VPN interface on each host and used the hypervisor to bind the tap to the emulated device. This created a logical link between interfaces on the emulated devices.

from our Tinc configurations in order to reduce overhead and complexity [46].

Adding or removing encryption in a particular instance of a topology is a trivial task and requires the modification of only three lines of code. We describe our use of templates in Section 3.2.4 and all Tinc configuration scripts are built from a template. Therefore in order to add encryption, removing three lines from the template will, by default, enable encryption.

Another requirement which differed from the original design of ERIK was the generation of scripts used for automation. The script generation methods used by ERIK were clear and simply designed but did not extend well to the requirements of multiple servers and larger topologies. We determined early that different design patterns were required to accomplish the distribution of devices and required a different underlying code structure.

3.2 DERIK

DERIK originated as an extension of ERIK. Though many of the concepts used in ERIK are also used here, the software structure is entirely different for DERIK and is depicted in

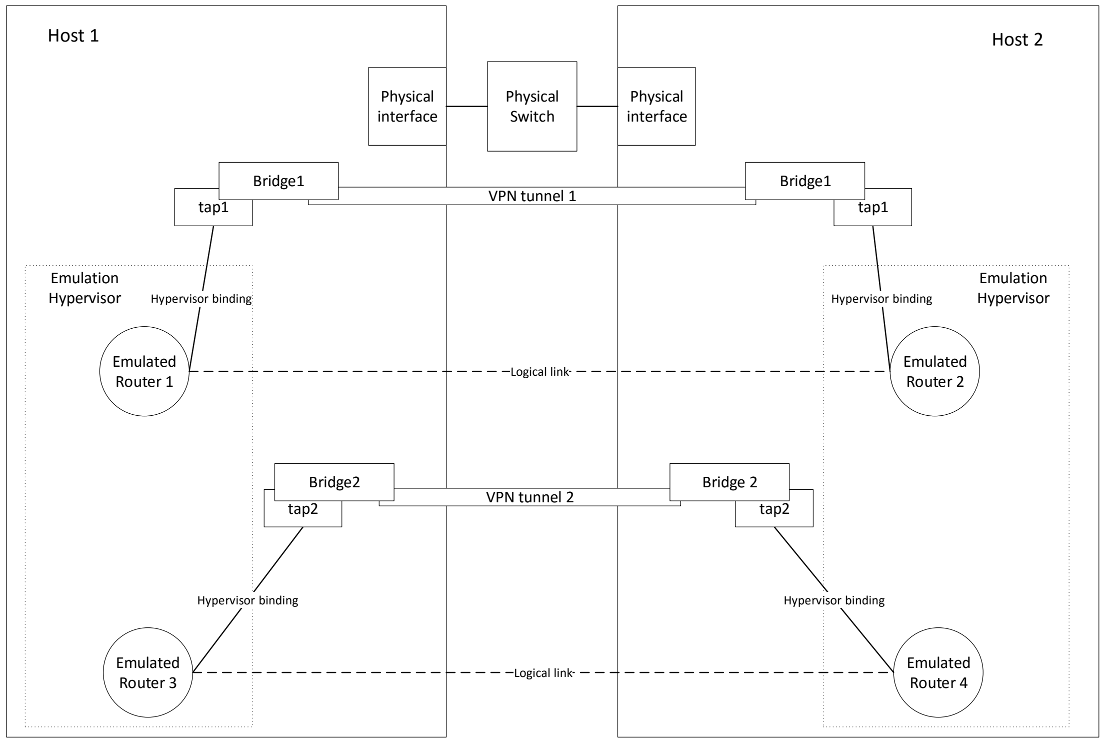

Figure 3.4: We manually configure Tinc to create multiple VPN connections between hosts for each link between emulated routers on different hosts

Figure 3.5. We segmented the topology creation process into five stages:

1. Creation of an abstract graph object: We use a custom graph class to define an abstract graph object and map a pre-made model.
2. Creation of an abstract emulated topology object: We define a topology as a set of devices and links and map a graph to said topology.
3. Distribution of emulated device objects to physical server objects: We use several methods to allocate emulated devices to physical hosts.
4. Creation of build scripts: we create configuration files for the emulated devices and scripts to automate interface creation and process execution for the physical hosts
5. Execution of build scripts: we use the local client to issue the commands required to execute the created scripts on the servers

We attempted to use as many software development best practices as possible during the

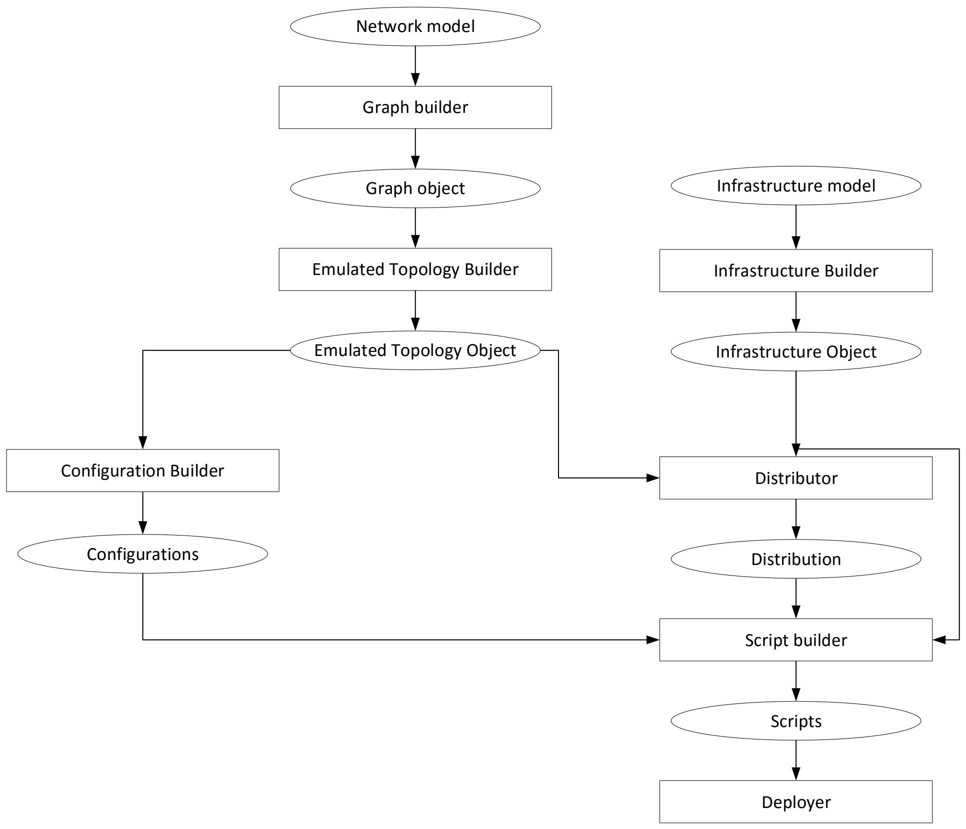

Figure 3.5: Builder classes take input and create new objects. The deployer class executes the created scripts on the remote servers.

creation of DERIK such as abstraction, modularity, extensibility, and code reuse. For example, in the instantiation of new device objects, we use the Factory Pattern [54] to dynamically create applicable device objects using a device factory. When the topology builder class maps a vertex, it checks the vertex metadata and recognizes the vertex represents a Cisco C7200 series router. The builder uses a generic Device Factory (instead of a special Cisco Device Factory) to create a new device. The factory understands that a new Cisco C7200 router is desired and creates such. Device objects inherit general attributes from a generic grandparent class, a more specific parent class, and then are instantiated as a specific model. For example, a Cisco C7200 series router inherits from a Cisco device, which inherits from a generic device, which inherits from an Autonomous System object. This greatly reduced the amount of repeated code and simplified the design of configuration generation. These design choices also improve the extensibility of the code, enabling the addition of other makes and models of devices in the future. If desired, another developer could simply add a Cisco 3700 series class, which inherits from the Cisco parent, and add the minor changes which differentiate it from a 7200 series. Additionally, we attempted to write highly granular, testable code and created tests which verify correctness for small blocks of code throughout much of the package.

3.2.1 Creation of a graph object

The graph creation functionality of DERIK provides an abstraction between third party packages and the emulated topology object. ERIK relied on the NetworkX [51] Python Package for model development and graph manipulation [28] and we also use NetworkX for the same purposes. In order to more loosely couple the topology building process to the model, we added the additional level of abstraction in the form of a graph object. This will provide flexibility if future users choose to use another package besides NetworkX.

A DERIK graph is modeled after a standard mathematical graph, defined as a set of vertices and edges:

$$G = (V, E)$$

where each vertex and edge is also an object. In the model, a vertex is representative of a router, and an edge is representative of a link connecting two routers. The graph itself, each vertex, and each edge include attributes which create a set of metadata about the entire graph object. Figure 3.6 is a Unified Modeling Language (UML) diagram of the graph, edge, and vertex classes. This UML diagram depicts the attributes which are inherited from the parent class, and additional attributes within the child classes which extend the parent.

DERIK's graph builder functionality takes a graph model as input in Gephi, GraphML, or GML file formats and uses the NetworkX Python package to read and construct a NetworkX graph object. It then maps the NetworkX graph object into a generic DERIK graph object and computes some additional derived attributes of the graph which can be later used for both graph and network analysis. The vertex and edge object also contain specific device or link data allowing network information such as AS number to be added within a model instead of created by the system.

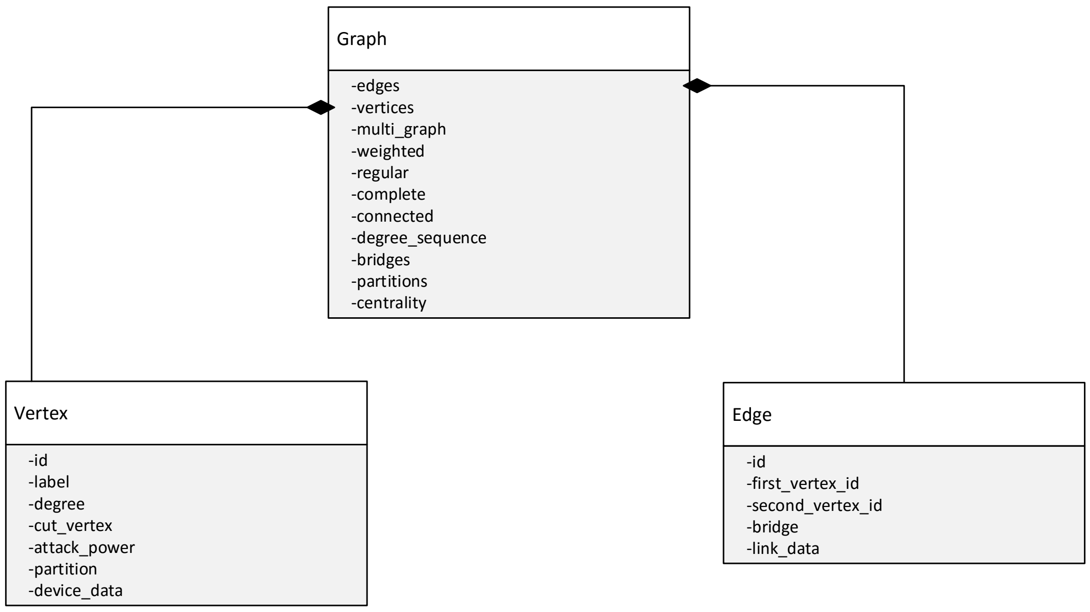

Figure 3.6: We consider a composition relationship between edges and nodes as they do not exists on their own, but exist within a graph.

3.2.2 Creating an emulated topology object

We chose to add another layer of abstraction into the system in order to simplify the distribution of a topology and increase the extensibility of the system. The topology builder class takes a DERIK graph object as input and maps the graph into an emulated topology object. For example, the topology builder class reads each vertex object and creates a device object based on the attributes of the vertex. There is a one-to-one correspondence between a vertex and a router, with a link between routers where there is an edge between devices. Figure 3.7 is a partial UML diagram demonstrating the relationship similarities.

By adding this additional level of abstraction, it allowed us to dynamically create configuration scripts for the device and hypervisor command scripts in a loosely coupled fashion. For example, in a real-world network two routers can connect to one another even if one is a Juniper router and one is a Cisco router. Both have interfaces, Internet Protocol (IP) addresses, and may have access lists; but the configuration syntax required to administer the routers may be quite different. By abstracting these physical devices into abstract device objects and separating them from the abstract graph model, we add flexibility to both our

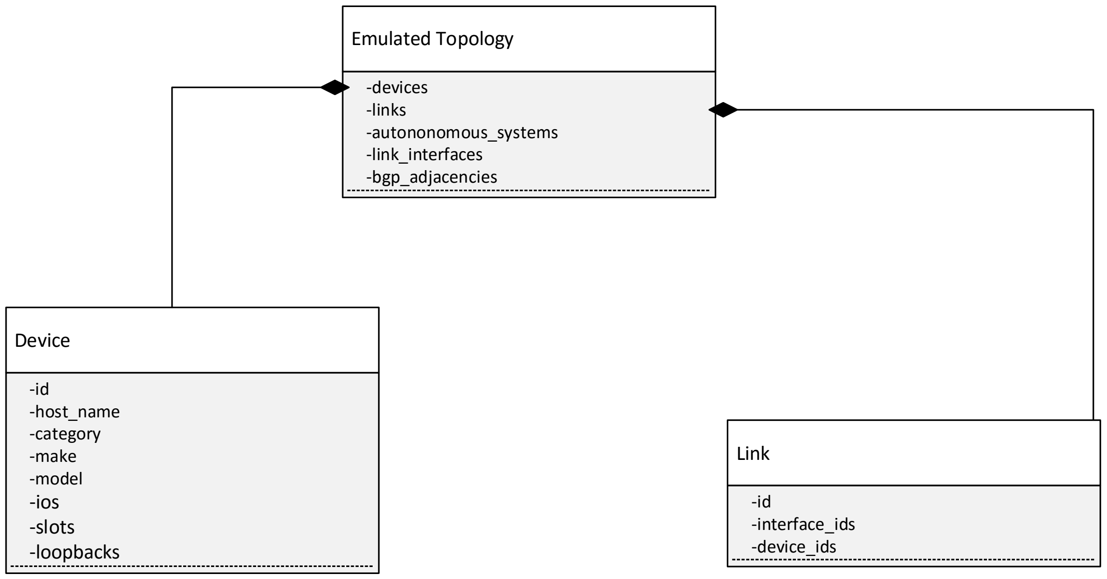

Figure 3.7: We consider a composition relationship between links and device as they do not exists on their own, but exist within a topology.

model input, and simplify the creation of configurations and build scripts later. In our implementation, we had our device class inherit from an Autonomous System class. This method allowed us to easily conduct BGP relationship analysis and apply it towards each device during the creation of device configuration files. Using this pattern allowed us to more easily apply desired routing policies which we will discuss in more detail in Section 3.2.4. Figure 3.8 shows the inheritance we used to simplify the instantiation of device objects.

In addition to mapping a graph, the emulated topology builder also creates BGP relationships within the topology based on the metadata provided from the model. This derived metadata is generic enough to transcend make and model, and is used later in the configuration building step. The overall software flow is depicted in Figure 3.5.

3.2.3 Distributing

In order to distribute the emulated topology, the distribution class takes in the emulated topology, available resources (as an infrastructure object), and the distribution objective as

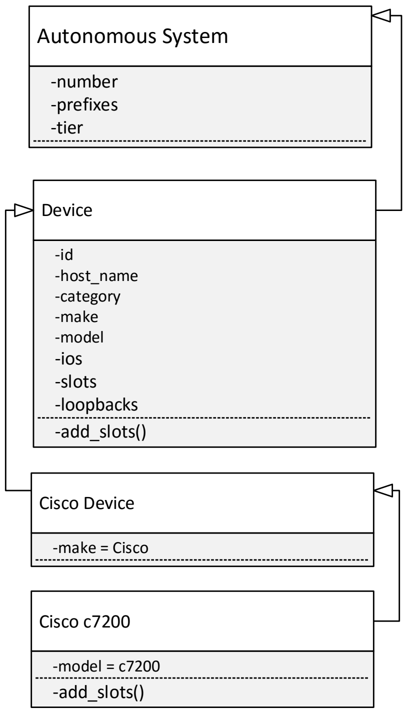

Figure 3.8: A device inherits from the Autonomous system class and specific makes and models of devices inherit further. Some methods are overloaded such as the "add-slots" method, and are specific to a model of device.

parameters. An infrastructure object is a representation of the available physical resources. For example, an infrastructure may consist of 20 servers, each of which is capable of supporting 10 emulated routers and physically connected by a physical switch. This is then modeled as a list of server objects with distinct attributes. The distribution calculation parameter indicates which method of distribution to use: uniform, heuristic, or min-load. The distributor then allocates emulated devices to physical servers and returns a modified infrastructure object, where the server attributes have been updated to indicate which devices will be allocated to each server.

Uniform distribution

We first considered a uniform distribution where devices are allocated to each server in a "round-robin" fashion. In this method, each server would be allocated one device before any server is allocated a second device. This pattern continues until all devices have been allocated, resulting in a distribution where all servers have an equal or nearly equal number of devices. Though a uniform distribution is simple and quickly calculated, ideally we desire to allocate the devices in as efficient manner possible for some given metric.

When the developers of ERIK placed a network on a server where the amount of memory allocated to the routers exceeded the amount of memory available on the physical server, the emulated devices began failing and behaving in unexpected manners. This behavior indicated that the bottleneck in the distribution would be the available memory on a given physical host. To overcome this bottleneck, we simply input an attribute of “capability” for each physical host into the infrastructure model where capability is a number indicating how many emulated devices we expect a server can accommodate. For instance, we initially allocated 512MB of memory to each device. If a server has 2048MB of free memory, we would input a capability of 4.$(4 * 512 = 2048)$ for the server. This is a manual indicator of the capacity of the machine and prevents the distributor from allocating more devices than the host is capable of supporting. In our cases we assume identical processing and memory requirements for each emulated router which reduces the complexity for distribution with regards to these characteristics. If one were to create a model where each individual router can have different memory and processing requirements, techniques such as “Bin Packing” could be an applicable alternative [55].

Optimal min-load distribution

We determined that the next bottleneck could be the physical interface on a physical host and a uniform distribution does not account for limits on the physical links between machines. With a large network, there is potential for many VPNs to traverse a single physical link, and therefore the potential to saturate the physical link, thereby impacting the emulated network. We therefore consider distributions of devices that optimize some criteria, such as: minimize the number of required virtual interfaces or minimize the amount of traffic across the physical link VPNs. We decided that generating a traffic model for a specific network was not within scope of this project and instead decided to use the metric of “edge betweenness centrality” as an indicator of potential traffic load. However, if a specific traffic model is known, it can be added to the network model as traffic load and used in place of our chosen edge betweenness. Rye et al. also used betweenness-centrality as a measurement to determine key links for fault analysis [28]. In references [28], [51], [56] link betweenness is defined as:

$$b(e_i) = \sum_{s,t \in V(G)} \frac{\sigma(s,t|e_i)}{\sigma(s,t)}$$

where $\sigma(s,t)$ is a geodesic in a graph G between vertices $s \neq t$, and $\sigma(s,t|e_i)$ is a geodesic between $s \neq t$ containing the edge e_i.

For example, consider the graph G depicted in Figure 3.9. If we have two available physical servers, each capable of supporting three emulated devices, an allocation for minimizing the sum of the load across the physical link between servers is depicted in Figure 3.10. If each router sends a single packet to every other router, we observe a total of 30 packets sent, 18 of which traverse the physical link. Alternatively, if we select a random distribution as depicted in Figure 3.11 we would have a total 35 packets traversing the physical link. This simple example illustrates the importance of thoughtful allocation in a potentially resource constrained environment.

The method we developed for optimizing allocation was through the use of a linear program, specifically a mixed integer program, expressed in NPS format as follows:

Indicies

r	router $r = 1, 2, ..., R$ where R is the number of routers in the topology
m	machine $m = 1, 2, ..., M$ where M is the number of physical machines available

Parameters

$T_{r,r'}$	The expected traffic between Router r and Router r' [B/s]
Max_m	The maximum number of routers machine m is capable of holding

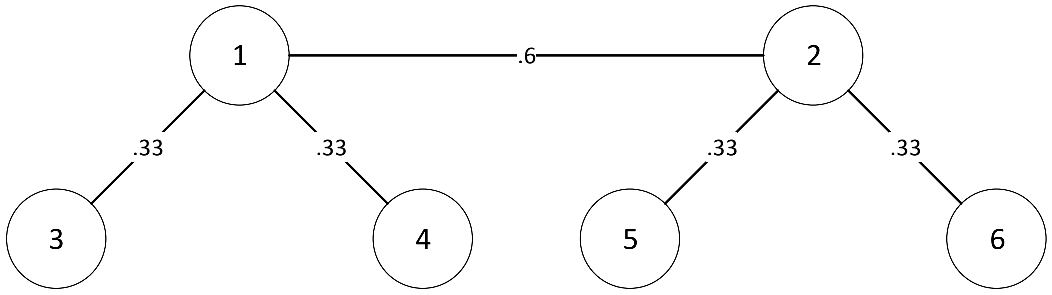

Figure 3.9: A simple graph model of six nodes connected by five edges. This could be a small example of two tier one providers (nodes 1 and 2) and four customers (nodes 3, 4, 5, and 6).

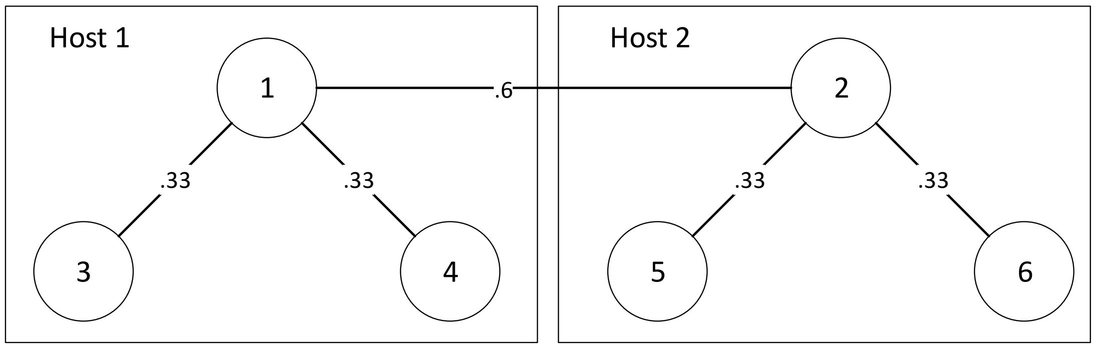

Figure 3.10: If load on the physical link is measured as the sum of the loads of the simulated links, then the "load" on the physical link is minimized.

Decision Variables

$R_{r,m}$	Router r is on machine m [binary]
$A_{r,r',m}$	Routers r and r' are not both on machine m [binary]

Constraints and objective function

Minimize the amount of traffic between routers on different physical machines.

$$\text{Minimize} \sum_{r,r',m} T_{r,r'} * A_{r,r',m} \forall r, r', m$$

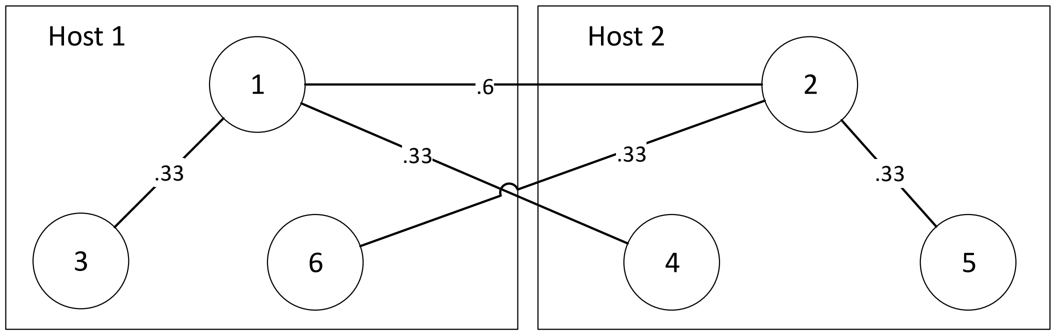

Figure 3.11: If load on the physical link is measured as the sum of the loads of the simulated links, then the "load" on the physical link is not minimized. A packet flowing from node 1 to node 6 will cross the physical link twice, despite both emulated devices running on the same physical machine.

subject to:

Cannot place more routers on a server than the server is capable of
$\sum_r R_{r,m} \leq \text{Max}_m, \forall m$

Each router can only be on one machine
$\sum_m R_{r,m} \leq 1, \forall r$

Model an 'and' constraint for two routers on different physical machines
$A_{r,r',m} \leq R_{r,m}, \forall r, r', m$
$A_{r,r',m} \leq (1 - R_{r',m}), \forall r, r', m$
$A_{r,r',m} \leq R_{r,m} + (1 - R_{r',m}) - 1, \forall r, r', m$

The program attempts to calculate the lowest sum, relative to all possible sums, of the load on physical interfaces. This is accomplished by multiplying the link load by the binary value indicating whether the devices incident to a given edge are on the same physical host, for all edges. Due to our use of Python for developing DERIK, our implementation consisted of converting this program into Pythonic form using the PuLP package [49]. As data, we used our available server infrastructure as described in Section 3.1, emulated devices, and generated a matrix from the link-betweenness values of our emulated topology. Our implementation first analyzes the server's capabilities and if the servers are incapable of supporting the specific number of devices, returns an error stating such. Additionally, if a single server can support all the devices, the solver immediately returns the trivial solution of all devices allocated to a single server. PuLP also installs the free CoIN-OR solver [49] by default which we then used to solve the linear program and provide an allocation of devices to servers. The results were stored as a Python dictionary, with the server id as a key, and a list of device ids as the value. This dictionary was then used to add a list of device objects to each specified server's attributes. As shown in Figure 3.5, this entire process is automated within DERIK.

A limitation with this approach is the sheer size of the linear program. Though simple in its expression, as the number of devices increases, the number of potential solutions increases exponentially. The problem itself is still linear, but the number of constraints increases as a square of the number of devices. This large size places a significant burden on the solver and increases both the memory requirements of the host running the solver software and the time required to complete. The solver may be able to find a feasible integer solution, and find a lower bound (on a minimization problem) which is not an integer, but will continue to run until it finds the optimal solution which can take hours or days depending on the size of the topology and the solver software used. Figure 3.12 depicts solves times using a Microsoft Surface Pro 4 and the CoIN-OR solver.

In an attempt to reduce the running time, we implemented another feature of linear programming, fractional gaps, as a configurable variable in the distribution method. Implementing a fractional gap allows the solver to return a feasible solution which is within a percentage of its known best possible answer, even if the optimal solution is not yet reached. This feature reduces the solve time but may return a sub-optimal, though feasible, solution. Additionally, it allows the solver to have a set time-limit for the solving process. Without the gap and time limit, a near-optimal solution may be found quickly, but the solver will continue to run as long as necessary to find the optimal solution.

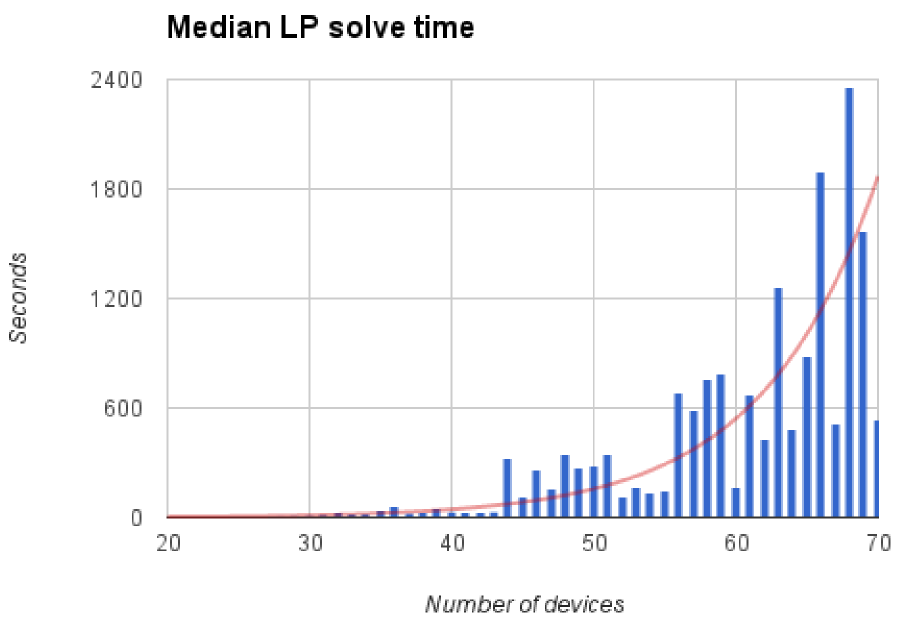

Figure 3.12: As the size of the graph increases, there is an exponential trend in the time it takes to solve the problem.

Heuristic distribution

In order to provide another, faster option for large topologies, we also developed a heuristic approach to distribution. This approach has a significant runtime improvement with a time complexity of $O(n \log n)$ where n is the number of edges in a graph G [57]. For our heuristic distribution, we simply sort the edges by the link-betweenness value and allocate both devices incident to said edge to a single server, until the max capacity of the server is reached. At which point, we begin allocating devices to the next server. This method is depicted in pseudo code in Algorithm 1.

3.2.4 Building scripts

Manually executing the creation and deployment of a topology becomes cumbersome as the size of the topology grows. We desired to automate the process as much as possible and developed DERIK to create a series of scripts which can be run automatically as part of the software flow. For example, if DERIK is deploying a topology to a Linux server, it will automatically create bash scripts that execute the commands to create tap interfaces and

Algorithm 1 Heuristic Distribution Pseudo code

```
topology.links.sort(key = link_betweenness )
current_server_index = 0
current_server = all_servers[current_server_index]
for link in topology do
    for device incident to link do
        if current_server.num_devices_allocated ≥ current_server.max_capability then
            current_server_index + = 1
            server = servers[current_server_index]
        server.add_device(node)
```

bridges. These scripts are built from templates and require no additional user interaction once a model has been generated and input into DERIK.

In order to minimize the dependency on a particular make or model of emulated router and to simplify the creation of device configuration scripts, we choose to use a templating engine, *Jinja2* in the creation of our build scripts and configuration files [48]. For example, the Script Builder takes a Device object (discussed in Section 3.2.2) as input and creates a configuration file based on a template for the particular make and model contained in the Device object's attributes. We further extended this by using template inheritance and nested templates in order to minimize file Input-Output (IO) and simplify the administration of templates. For example, many Cisco operating systems contain similar sections, syntax, and commands for a baseline router configuration so we create a base "Cisco" template. If a model "c7200" router has a slightly different syntax for access-list creation than model "Nexus 7000", we can simply override the parent access-list template by creating a new "nexus-7000" directory containing a modified access-list template. The configuration builder analyzes a device object's attributes, including make and model, and selects the appropriate templates to use for the generation of the configuration file. Each template contains fields which correspond to dynamic information required by the configuration but specific to each instance of a device. For example, a baseline configuration may contain the field `"hostname"`, which is then populated with the device's attribute, `"hostname"`.

The configuration builder class uses the templates, device object attributes, and derived data from the emulated topology to create the configuration files for each device. The derived data computed by the topology builder class is necessary for specific configurations such as BGP neighbor commands. The authors of ERIK choose to implement some standard traffic shaping policies in BGP configuration such as preferring customer traffic over peer or provider; in order to generate these configurations automatically, some derived attributes needed to be calculated for relationships between nodes. One significant difference from ERIK we implemented is the separation of configurations into their own files instead of encoding them into base64 and pushing them via the hypervisor commands. As the number of links per router increases, the number of BGP neighbors increases and the total size of the configuration increases. Dynamips has a limit on the size of the message that can be sent via its console input and we quickly exceeded that size limit [52]. Instead, we used another Dynamips command *vm set_config* in order to provide a location for a configuration file stored on the server.

We also use the templating methodology for the creation of scripts which will run on the physical hosts. For example, scripts which start and configure the hypervisor daemons and VPN daemons are dynamically created based on the type and number of physical hosts, emulated devices, and inter-connections required. All the scripts and configurations are then saved locally in order to provide test repeatability and for later analysis.

3.2.5 Execution of build scripts

The deployment of an emulated topology to the physical servers is executed from the local client and can be broken into four sequential steps, each dependent on the prior steps:

1. Directory structure creation
2. Transfer of scripts and configuration files
3. Creation of virtual interfaces
4. Creation of emulated devices

In order to automate the execution of the scripts we used the Python package "Fabric" [47]. Fabric provides an interface for automating SSH functionality using the Python programming language. In our case we used it to automate the transfer of files created in section 3.2.4 to remote servers and the execution of commands on the remote servers. For example, as described in section 3.2.4 we create a script which sends console commands to a hypervisor. Fabric is used to push the script to the remote server and then automates the execution of the script at the appropriate time.

Dynamips uses a telnet session to receive the commands necessary to create or modify new devices. Automating this process proved challenging due to the “nested” sessions required, and we decided to use the UNIX “expect” utility to execute pre-created scripts. In order to parallelize the process as much as possible, we create separate scripts for each hypervisor which can be started in parallel by a bash script, which in turn is started as part of the deployment process by a remote command issued from Fabric. With the potential for many instances of hypervisors running on a server simultaneously, and each hypervisor script containing up to 1000 commands, this greatly reduced the run time for the deployment step versus running in sequence.

CHAPTER 4: Results and Analysis

In this chapter we first discuss the results obtained creating emulated topologies using DERIK. We then detail the analysis we conducted on the methods used to distribute the topologies. Finally, we examine the execution of a BGP hijack scenario we conducted as an example of the utility of DERIK as an emulation platform for large topologies.

4.1 Analysis of DERIK Topologies

Our initial goal was to verify that DERIK was able to create a topology of over 1000 emulated devices running on two physical servers. This would create an emulated network over three times larger than Rye et al., and meet our objective of distributing the topology across physical machines.

We first analyzed a topology created by DERIK using a random Barbasi-Albert model as input where each node represented a single router and single AS. As input into the NetworkX random graph generator, we provided an order of 1001 and an attachment value of 3, meaning each new vertex would preferentially attach to three random, existing vertices. The attachment value indicates how many edges are created between a node added to the graph, and the existing nodes. The NetworkX graph generator function takes an integer as the attachment value. The resulting graph is sensitive to this attachment parameter; minor changes can cause large differences in the total number of graph edges. Too few edges in the model and the resulting network was sparse and not representative of the Internet structure desired. Alternatively, a large number of edges within a graph created high degree nodes that we are unable to emulate due to restrictions on the total interface count on a single emulated router. For example, a Cisco 7200 series routers can support six network modules, each with eight interfaces, for up to 48 interfaces per device. Relaxing the one-to-one mapping of device to AS is a valuable exploration for future work and is discussed in Chapter 5.

In order to reduce the size of the graph after creation, we iterated through each edge and randomly deleted each with a probability of .25 which resulted in a total size of $2,280$

edges. In order to approximate the structure of the Internet, we divided our model into three tiers using similar parameters to those described by Rye et al. [28], though we were able to reduce the total percentage of Tier 1 nodes due to the larger size of the network. The topology was partitioned into 20 Tier 1, 300 Tier 2, and 681 Tier 3 ASs.

To ensure that the model was representative of our desired tiered structure, we then analyzed the entire graph to ensure it met five characteristics:

1. All Tier 1 nodes had at least one Tier 1 neighbor.
2. All Tier 2 nodes had at least one Tier 1 neighbor.
3. No Tier 3 peers existed.
4. The maximum number of neighbors for any device did not exceed the maximum number of ports the device possessed.
5. The graph consisted of a single connected component

If any of these characteristics were not met, our model builder automatically created or deleted an edge to adjust. When deleting an edge, the model builder verified that the graph was not disconnected by the edge removal. If the graph was disconnected, the builder randomly selected a node from each component, verified that it was not breaking the requirements by adding a tier 3 peering, and added the edge. When adding an edge, it randomly selected two nodes which met the requirements (i.e., a tier one node with no other tier 1 neighbors, and then another tier 1 node), and added an edge between the two. The model builder then repeated the analysis until all requirements were met. The final, emulated network consisted of 1001 Cisco 7200 series routers, connected via /30 networks, each advertising a /24 network in BGP as an AS. This specific model is not meant to be representative of any particular network, but instead was meant to use a common model development method to test the limits of our platform.

We created and deployed this network using DERIK and the heuristic method of distribution to two physical servers: 750 emulated routers to Server A, and 251 routers to Server B. The min-load distribution method for this size network took prohibitively long to execute; we defer using min-load on very large networks to future research. The heuristic distribution results in 20 Tier 1 nodes, 300 Tier 2 nodes, and 430 Tier 3 nodes on Server A; and 0 Tier 1 nodes, 0 Tier 2 nodes, and 251 Tier 3 nodes on Server B. The physical hosts were connected to one another by a single physical gigabit Ethernet interface with a physical switch between. The percentage capability of each machine was predetermined by using the available memory on the server used by Rye et al. in their 300 node test, and extrapolating upward in a linear measurement relative for our servers [28]. These capabilities were relative to one another and the size of the topology, not the max capability of the machines. We will discuss the server's full capabilities in our next example. The resulting distribution created 617 VPNs (and the applicable virtual interfaces) on each machine.

The total process completed in less than 15 minutes with over 99% of the time used for execution of the remote commands to each of the servers. Much of the time required is due to the number of individual commands run to create the virtual interfaces. Due to the security configuration on our servers, we were required to execute each command remotely. This process could be expedited by creating and executing a single script including all commands, on each server. This was completed on a Microsoft Surface Pro 4, connected via a wireless connection to the network.

To verify successful deployment, we conducted both manual analysis and automated analysis. We first established a remote session with each of the physical servers and verified that the Dynamips and Tinc processes were running. We then consoled into a randomly chosen router and viewed the BGP routes to verify that the route table was populated. Alternatively, in Section 4.3 we discuss connecting the topology to the BIRD daemon which provides an automated way to view the number of BGP routes learned by each device and can be verified against the number of expected routes. Finally, we conducted some traffic analysis as described in Section 4.2 to verify that traffic was successfully traversing the physical interfaces.

4.2 Analysis of Optimization

In order to test the efficiency of our distribution methods we devised a simple traffic scenario to determine the amount of traffic traversing the physical interfaces of our physical hosts. We used tcpdump to monitor and record packets traversing the physical interface on each server. Tinc uses pre-defined ports for each of the VPN connections and we applied tcpdump filters to only capture packets on the ports configured for the VPNs. We chose to use Internet Control Message Protocol (ICMP) traffic rather than something more complex, such as a file transfer, to establish a lower bound and to ensure we did not immediately overwhelm any point in the paths. Each ICMP query and reply also contains a sequence number, which simplifies analysis of loss.

We developed a simple script in which each emulated device pinged every other device using the "`send_con_msg`" command in the Dynamips hypervisor. The script was built as part of the script building process described in section 3.2.4, pushed to the respective servers during deployment, and then executed manually via SSH. We chose to send five, 100 byte requests; and randomized the order of the destination addresses for each device in order to prevent all devices from pinging a single destination simultaneously. Additionally, the script started each ping session with a delay of $\frac{1}{n}$ seconds, where n is the number of sessions. This delay reduced simultaneous bursts of traffic throughout the scenario. An individual script was created for every hypervisor instance and run in parallel resulting in approximately $\frac{m}{5}$, where m is the number of devices, simultaneous ICMP requests and replies occurring on the servers at any given time.

To compare optimization methods, we: i) created topologies of 50, 100, and 150 devices; ii) distributed the topology using a uniform (random) distribution, heuristic distribution, and min-load distribution; and iii) executed the traffic analysis methods described above on each individual scenario. We discovered a significant decrease in the amount of traffic observed on the physical link using either a heuristic distribution or a min-load distribution versus a random distribution. We chose not to conduct comparisons of topologies larger than 150 nodes as the linear program run time for the min-load method exceeded our testing time limits. Figure 4.1 depicts the results.

Note that the uniform distribution caused more traffic to pass through the physical interfaces in all topologies analyzed. This is due to a larger number of paths traversing the physical interface multiple times, as depicted earlier in Figure 3.11.

Additionally, we analyzed the captured traffic for loss to determine the amount of loss given a low traffic load scenario. As we originated the ground truth traffic, we knew how many source/sink ICMP packets should be traversing the link. We were thus able to programmatically analyze the capture to determine how many packets, and from which sources/destinations, were lost. For example, we know that five ICMP echo requests should exist for a (source, destination) tuple. We also know that five ICMP echo replies should exist for the reverse (destination, source) tuple. We tested on 50, 100, 150, and 300 node networks

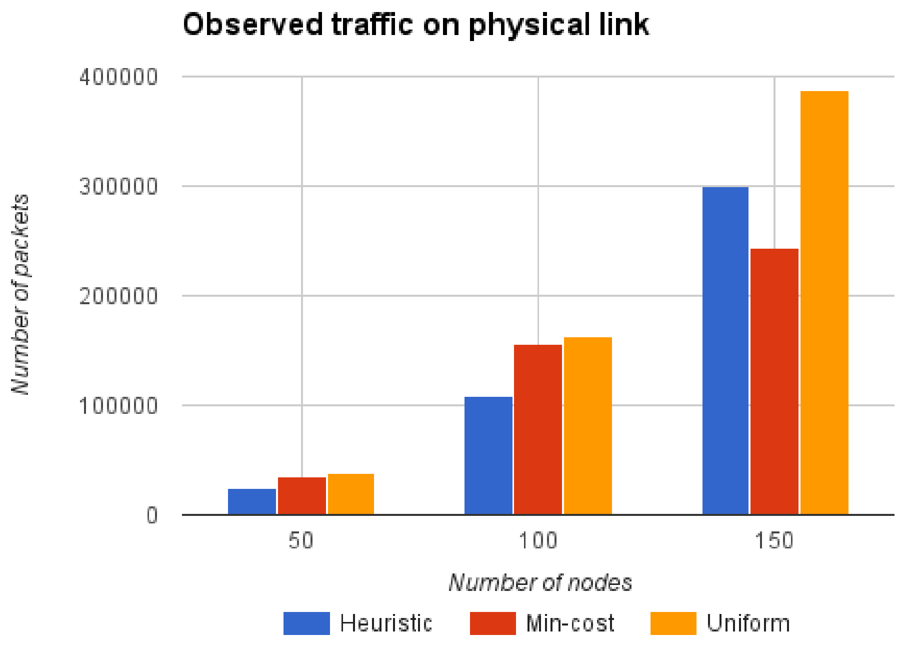

Figure 4.1: The amount of traffic is reduced using both the heuristic method or min-load method.

and our captures indicated 0 packets lost. As we scaled up the size of our network, we began losing a small percentage of packets. For example, with a 1001 node network, we detected a loss of 88 of 6347020 packets or .00002% of packets. We had collected packets on the physical interfaces of both hosts and our captures contained identical results, indicating that the loss was not due to physical restrictions on the host, but rather to unresponsive emulated devices at some point in the path. Though we were unable to identify the root cause of the unresponsiveness, we speculate it may be caused by delayed route propagation; or the overwhelmed memory or processor of an emulated device; and believe it worth further research. In Section 5.1 we discuss some opportunities to address loss in these scenarios.

4.3 Proof of Concept Usage

We desired to demonstrate the utility of DERIK by executing a real-world, non-trivial application and analyzing the results to understand its effects. As stated in Section 2.4, BGP is a challenging protocol to simulate and we desired to demonstrate the value of an emulation platform by developing a BGP hijacking scenario.

A BGP hijack attack is the false advertisement of a prefix which is actually owned by another AS [58]. This can occur on accident by misconfiguration, or by an adversary for malicious purposes. These attacks, whether accidental or nefarious, can cause serious effects on Internet availability. A hijack could take multiple forms as indicated in [58], such a falsely advertising a connection to another AS or falsely advertising a prefix it does not own. We chose to execute an attack where the *hijacking* AS falsely advertises a prefix owned by the *hijacked* AS.

Using an emulated topology to execute this attack scenario provides several interesting insights. Implementing policies on each router (and therefore each AS) using actual Cisco configurations, allows us to observe how these configuration changes directly impact the effects of a hijack. These changes will directly affect the percentage polluted, rate of pollution, and parts of the topology polluted due to filtering of the routes advertised between neighbors. The differences between a topology where no policies are implemented, and a topology containing policy can provide insight which may be missing from a simulated environment.

To conduct this scenario, we slightly extended DERIK to allocate an unused interface on each of the individual routers and bind it to a BIRD [59] daemon running on the host using Tap and Bridge interfaces. Binding to unused interfaces is one way in which we anticipate users extending and connecting to a DERIK emulated network.

BIRD is a routing daemon which can import and export BGP routes when connected to another emulated router, or to the kernel of a Linux machine [59]. We extended DERIK to automate the creation of a BIRD configuration file to allow it to receive BGP updates, but not advertise any prefixes of its own. DERIK then automatically configured a BGP neighbor pairing for each device and the AS advertised by the BIRD configuration. By doing this we created a simple "looking glass" for every device in the topology using a single daemon on the host. Figure 4.2 depicts the structure we used. BIRD provides the added benefit of logging to a Multi-Threaded Routing Toolkit (MRT) dump file, which we could analyze using a custom Python script.

We then used the same "`send_con_msg`" feature of Dynamips to enable us to automate

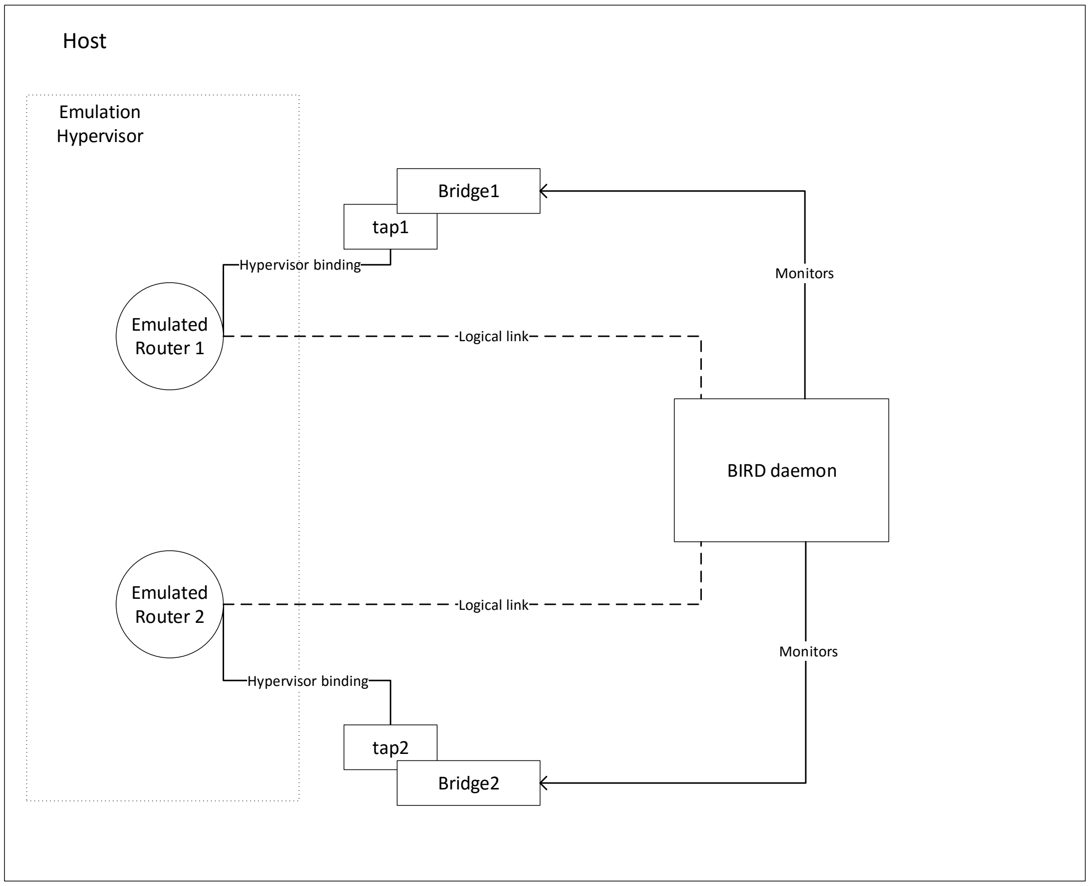

Figure 4.2: BIRD monitors the Bridge interfaces which allow it to receive BGP updates as a neighbor of the emulated routers. Each device is bound to a unique bridge interface and BIRD pairs using the required address and AS number as configured.

scenarios via a script such that different prefixes are hijacked at various locations within the topology. For example, a Tier 1 node could hijack a prefix already advertised by a Tier 2 node. For this scenario, we randomly selected a Tier 1 node as the attacker and a Tier 2 node as the victim. By modifying the configuration of the Tier 1 node to advertise the prefix currently advertised by the Tier 2 node, the topology was polluted and some other nodes now believed the Tier 1 node owned the prefix. For our purposes, the random selection of nodes was sufficient for demonstration, but the precise selection based on characteristics of the topology or nodes is another valuable method that could be considered in the future.

We executed a single hijack after waiting for the BGP paths to fully converge. The BIRD daemon provides an interface where we were able to see the number of advertisements provided by each node. Once the count reached the expected value, we knew the paths were converged. We then executed the attack and waited for the results of the hijack to converge using the same methods above. Once the polluted paths had converged, we stopped the BIRD daemon and saved the dump file for analysis. After removing the hijacked prefix, we allowed the network to re-converge before executing the next scenario. We analyzed the dump file by modifying scripts created by Jon Oberheide [60] which utilized the dpkt package [61]. The pseudo code outlining our method to find a polluted path update is shown in Algorithm 2.

We created a small topology of 50 nodes using the model building method described in Section 4.1. We chose a smaller topology to demonstrate the extensibility of DERIK and for ease of analysis. This resulted in five Tier 1 nodes, 15 Tier 2 nodes, and 30 Tier 3 nodes. Our topology was built using templates which included the policies described previously in Section 3.2.4. For example, a provider prefers customer traffic over peer traffic. We executed nine scenarios using the methods described earlier such as a Tier 1 node hijacks a prefix owned by another Tier 1 node, then a Tier 2 node, etc. We analyzed the dump files and charted the effectiveness of a node within a particular tier hijacking nodes within each other tier. The results are displayed in Figure 4.3. Additionally, the updates from the dump file include a time stamp so we were able to analyze timing information, such as convergence time, for a particular scenario.

Using DERIK to conduct these scenarios provided us with several benefits:

1. We were able to conduct the hijack using the actual behavior of Cisco routers running BGP and were not required to develop any special simulations.
2. We were able to configure the routers running BGP using the same configurations used by actual routers. For example, using a route-map to prefer traffic from customers over peers.
3. We were able to conduct relative time comparisons between BGP updates and determine the time taken for hijacks to converge.
4. We discovered that due to our implemented policies, certain hijacks had less effect on the flow of traffic.

Algorithm 2 Find polluted paths

```
paths = []
for update in updates do
    if update.prefix == attack_prefix then
        paths.append(path)
true = []
polluted = []
for path in paths do
    if last_as_in_path == attacker _as then
        polluted.append(path)
    else
        true.append(path)
polluted_ases = set()
for polluted_path in polluted do
    for true_path in true do
        if first_as_in_polluted_path == first_as_in_true_path then
            if len(first_as_in_polluted_path) $\leq$ len(first_as_in_true_path) then
                polluted_ases.add(first_as_in_polluted_path)
```

Consider the example graph depicted in Figure 4.4. Using no policies, if node 10 hijacks a prefix owned by node 1, both nodes 11 and 12 would be polluted due to a new shortest path advertisement from the hijacking node 10. The common policies implemented in our network prevent this from occurring by filtering route advertisements between Tier 2 peers. Therefore, if node 10 hijacks a prefix owned by node 1, neither nodes 11 nor 12 would be polluted due to a preferred path via their providers. These types of commonly implemented policies impact the behavior of many models and demonstrate the value of emulation.

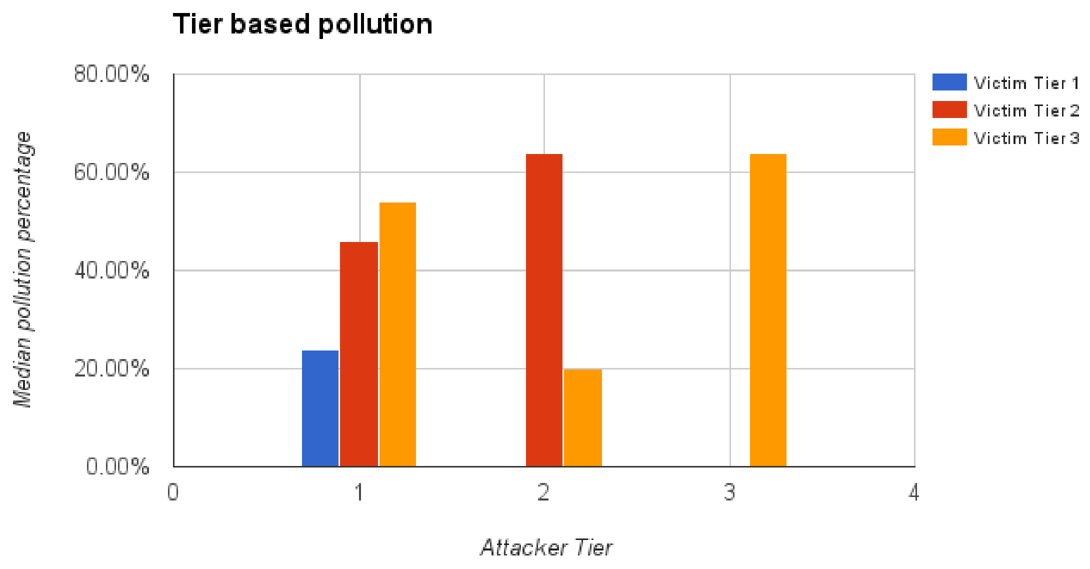

Figure 4.3: The x axis indicates which tier attacked (or hijacked) and each bar represents the victim tier. Note that the nodes we tested were ineffective in hijacking nodes within a lower tier.

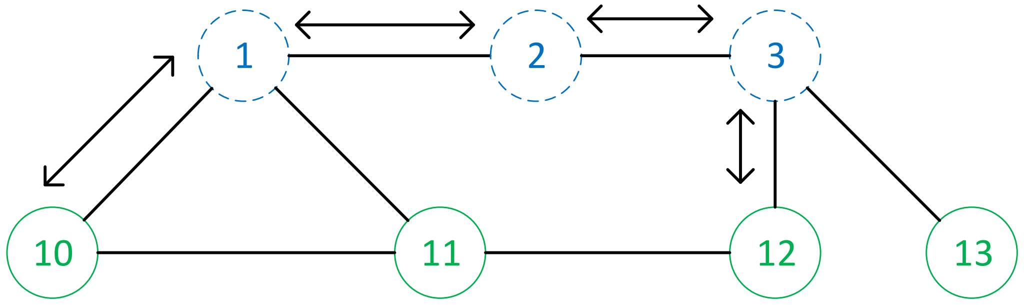

Figure 4.4: Nodes 1, 2, and 3 are Tier 1 nodes. Nodes 10, 11, 12, and 13. The standard behavior of BGP is overridden by policy and therefore impacts the effects of a hijack.

CHAPTER 5: Conclusions and Future Work

In this thesis we extended the ERIK platform in order to:

1. Automate the creation of an emulated network topology across multiple physical servers
2. Efficiently allocate emulated devices to minimize traffic between physical devices

The resulting system, DERIK, uses the following methodology:

1. Accept a network model as input
2. Analyze and distribute the topology to physical devices
3. Generate the automation scripts and configuration files
4. Deploy the topology from a local client to the physical infrastructure

Once the emulated topology is created, it can be utilized for arbitrary test scenarios of non-trivial topologies. In addition, we show how test scenarios can interact with the applications on the servers, connect to other Virtual Machines (VMs) running on the server, or pass modification messages via the hypervisor.

While we do not explore the upper-bounds of the topology size that DERIK can emulate, this thesis creates a topology of over 1,000 emulated routers (approximately three times larger than previously achieved by ERIK).

A primary contribution of this thesis is exploring the means by which emulated resources are distributed. Chapter 4 discusses the results of our optimization routines. We see a reduction in traffic between physical servers using either the heuristic approach or the min-load method versus a uniform allocation. These results indicate that simple consideration of placement will allow the system to best accommodate input topologies for a given physical infrastructure.

Additionally, we examine the benefits of emulation by creating a BGP hijack attack on our emulated environment. The execution of this experiment, as explained in Chapter 4, demonstrates that behavior is present in an emulated environment which may be missing, or require additional complexity to implement, in a simulated environment.

5.1 Future Work

In this section, we detail our suggestions for work that builds on our results and analysis.

Internal BGP

In our models we represented each AS using a single router in order to maximize the total size of the AS-level topology created. This method places several constraints on our networks. First, as the size of the network increases, our models will be limited in the number of links connected to a single device and will therefore limit the connections of a single AS. Analysis of BGP adjacencies from the Internet [42] show that many ASs have many more adjacencies than can be supported by a single physical device, and in order to model these, multiple routers per AS are required. DERIK currently only supports External Border Gateway Protocol (eBGP) neighbors and configuration for iBGP would be required for these models. We believe that adding this capability would allow the exploration of even larger topologies or models with larger degrees per AS.

Mixed device topologies

While our results from Chapter 4 were obtained using a network of entirely Cisco devices, additional emulation tools such as *Qemu* and *Quagga* exist and are widely used. We considered extensibility throughout the development process and simplified the addition of new platforms to the software, but did not implement devices other than Cisco models. Many real world networks are very heterogeneous and it would be beneficial to add the capability to create topologies consisting of multiple vendors, in order to analyze differences between vendor implementations of protocols.

During our distribution methods we consider all devices to require an equal amount of memory and processing. As discussed in Section 3.2.3, distribution of devices with unique memory and processing requirements is possible with some minor modifications. This could prove valuable for customizing specific nodes within a topology. For example, a Tier 1 node with a high degree could be allocated more memory within the hypervisor, or be configured to allocate more memory towards IO to account for an expected higher load.

Commercial solver software

Our attempts at linear optimization were completed using freely available solver software which may not be the ideal platform for a specific problem. Paid solver software such as IBM's CPLEX software [62] exists and may be beneficial for solving the min-load on larger topologies. Though the heuristic method performed very well, a guaranteed optimal solution for min-load may be desired or analysis of the linear program performance using alternative solvers would be beneficial.

Further experimental scenarios

We believe there are many opportunities for further scenario testing using DERIK. For example, the Department of Defense (DoD) and its subordinate agencies operate and control very large networks. Emulating portions of these networks before applying configuration or policy changes could prove valuable in optimizing performance and minimizing downtime. Our use of templates could also provide a benefit to the administration of tactical type networks. Currently, the use of a "baseline" configuration for devices is used to provide required elements such as access-lists. Then, much of the remaining configuration is completed manually to fit a defined network model. Automated generation of configurations using templates could reduce preparation time and time required for troubleshooting configuration errors.

As demonstrated by Rye et al., analysis of measurement tools is another valuable research opportunity. New and innovative measurement tools, such as BGPStream [63], are developed but require testing and validation before they are employed in the real world. We believe DERIK is a useful platform for creating a ground truth network to test such tools against.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX: [Python Code for Linear Program]

```
@staticmethod
def _min_load_distribution(emulated_topology, servers):

    distribution = {}
    TOLERANCE = .00001

    num_servers = len(servers)
    num_devices = len(emulated_topology.devices)

    # create a mapping of position vs id
    server_mapping = {idx: server.id for (idx, server) in enumerate(servers)}
    device_mapping = {idx: device.id for (idx, device) in enumerate(emulated_topology.
        devices)}

    server_idxs = range(num_servers)
    device_idxs = range(num_devices)

    traffic_matrix = Distributor._populate_traffic_matrix(emulated_topology.links,
        num_devices)

    max_num_devices = [server.max_devices for server in servers]

    if sum(max_num_devices) < num_devices:
        raise RuntimeError("The␣servers␣cannot␣support␣this␣many␣devices")

    elif max(max_num_devices) >= num_devices:
        warnings.warn("All␣the␣devices␣will␣go␣on␣one␣server...")

    setup_time_start = time.time()

    allocated = LpVariable.dicts("Allocated", [(s, d) for s in server_idxs
                                               for d in device_idxs], 0, 1, LpBinary)

    server_cost = LpVariable.dicts("ServerCost", [s for s in server_idxs], lowBound=0)

    inter_server = LpVariable.dicts("Inter-Server", [(s, d1, d2)
                                                     for s in server_idxs
                                                     for d1 in device_idxs
                                                     for d2 in range(d1, len(
                                                         device_idxs))], 0, 1,
                                                         LpBinary)

    problem = LpProblem("RouterAllocation", LpMinimize)

    for d in device_idxs:
        problem += lpSum(allocated[(s, d)] for s in server_idxs) == 1
```

```
for s in server_idxs:
    problem += lpSum(allocated[(s, d)] for d in device_idxs) <= max_num_devices[s]

for s in server_idxs:
    for d1 in device_idxs:
        for d2 in range(d1, len(device_idxs)):
            problem += inter_server[(s, d1, d2)] <= allocated[(s, d1)]
            problem += inter_server[(s, d1, d2)] <= (1 - allocated[(s, d2)])
            problem += inter_server[(s, d1, d2)] >= \
                       allocated[(s, d1)] + (1 - allocated[(s, d2)]) - 1

problem += lpSum(server_cost[s] for s in server_idxs) >= -1

problem += lpSum(inter_server[(s, d1, d2)] * traffic_matrix[d1][d2] for s in
     server_idxs for d1 in device_idxs
                  for d2 in range(d1, len(device_idxs)))

max_seconds = const.MAX_MINUTES * 60

problem.solve(PULP_CBC_CMD(msg=1, maxSeconds=max_seconds, fracGap=const.MAX_GAP))

if str(LpStatus[problem.status]) == "Infeasible":
    raise RuntimeError("Distribution␣infeasible")
elif str(LpStatus[problem.status]) == "Not␣Solved":
    if value(problem.objective) <= 0:
        raise RuntimeError("No␣feasible␣solution␣found")

for s in server_idxs:
    if s not in distribution:
        distribution[s] = []

    distribution[s] += [d for d in device_idxs
                              if allocated[(s, d)].varValue > TOLERANCE]

server_id_device_id = {}

for idx, sever_id in server_mapping.iteritems():
    server_id_device_id[sever_id] = []
    for device_idx in distribution[idx]:
        server_id_device_id[sever_id].append(device_mapping[device_idx])

return server_id_device_id
```

List of References

[1] L. Breslau, D. Estrin, H. Yu, K. Fall, S. Floyd, J. Heidemann, A. Helmy, P. Huang, S. McCanne, K. Varadhan *et al.*, "Advances in network simulation," *Computer*, no. 5, pp. 59–67, 2000.

[2] R. Y. Rubinstein and D. P. Kroese, *Simulation and the Monte Carlo method*. John Wiley & Sons, 2011, vol. 707.

[3] L. Costantino, N. Buonaccorsi, C. Cicconetti, and R. Mambrini, "Performance analysis of an lte gateway for the iot," in *World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2012 IEEE International Symposium on a*. IEEE, 2012, pp. 1–6.

[4] J. P. Rohrer, "Network modeling and analysis: Overview of modeling types," 2016, unpublished lecture.

[5] P. Gill, M. Schapira, and S. Goldberg, "Modeling on quicksand: Dealing with the scarcity of ground truth in interdomain routing data," *ACM SIGCOMM Computer Communication Review*, vol. 42, no. 1, pp. 40–46, 2012.

[6] H. Arbabi and M. C. Weigle, "Highway mobility and vehicular ad-hoc networks in ns-3," in *Proceedings of the Winter Simulation Conference*, 2010, pp. 2991–3003.

[7] M. Stoffers and G. Riley, "Comparing the ns-3 propagation models," in *Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2012 IEEE 20th International Symposium on*. IEEE, 2012, pp. 61–67.

[8] MATLAB, *version 9.0 (R2016a)*. Natick, Massachusetts: The MathWorks Inc., 2016.

[9] R. Bellman, "On a routing problem," DTIC Document, Tech. Rep., 1956.

[10] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley, "Large-scale network simulation: How big? How fast?" in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*. IEEE, 2003, pp. 116–123.

[11] ns-3. (n.d.). ns-3. [Online]. Available: https://www.nsnam.org/. Accessed Mar. 31, 2016.

[12] T. Feng, "Implementation of bgp in a network simulator," M.S. thesis, Computing Science, Simon Fraser University, Burnaby, British Columbia, 2004.

[13] J. Karlin, S. Forrest, and J. Rexford, “Autonomous security for autonomous systems,” *Computer Networks*, vol. 52, no. 15, pp. 2908–2923, 2008.

[14] M. Wojciechowski, “Border gateway protocol modeling and simulation,” M.S. thesis, Computer Science, University of Warsaw, Warsaw, Poland, 2008.

[15] C. Zhu, O. W. Yang, J. Aweya, M. Ouellette, and D. Y. Montuno, “A comparison of active queue management algorithms using the opnet modeler,” *Communications Magazine, IEEE*, vol. 40, no. 6, pp. 158–167, 2002.

[16] S. G. Thorenoor, “Dynamic routing protocol implementation decision between eigrp, ospf and rip based on technical background using opnet modeler,” in *Computer and Network Technology (ICCNT), 2010 Second International Conference on*. IEEE, 2010, pp. 191–195.

[17] E. Weingärtner, H. Vom Lehn, and K. Wehrle, “A performance comparison of recent network simulators,” in *Communications, 2009. ICC’09. IEEE International Conference on*. IEEE, 2009, pp. 1–5.

[18] B. P. Swenson and G. F. Riley, “Simulating large topologies in ns-3 using brite and cuda driven global routing,” in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 159–166.

[19] S. Nikolaev, P. D. Barnes Jr, J. M. Brase, T. W. Canales, D. R. Jefferson, S. Smith, R. A. Soltz, and P. J. Scheibel, “Performance of distributed ns-3 network simulator,” in *Proceedings of the 6th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2013, pp. 17–23.

[20] K. Renard, C. Peri, and J. Clarke, “A performance and scalability evaluation of the ns-3 distributed scheduler,” in *Proceedings of the 5th International ICST Conference on Simulation Tools and Techniques*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), 2012, pp. 378–382.

[21] S. Nikolaev, E. Banks, P. D. Barnes Jr, D. R. Jefferson, and S. Smith, “Pushing the envelope in distributed ns-3 simulations: One billion nodes,” in *Proceedings of the 2015 Workshop on ns-3*. ACM, 2015, pp. 67–74.

[22] S. H. Y. Rekhter, T. Li. (2006, January). A Border Gateway Protocol 4: BGP4. [Online]. Available: https://tools.ietf.org/html/rfc4271

[23] S. Rampfl, “Network simulation and its limitations,” in *Proceeding zum Seminar Future Internet (FI), Innovative Internet Technologien und Mobilkommunikation (IITM) und Autonomous Communication Networks (ACN)*, 2013, vol. 57.

[24] Packet Tracer. (n.d.). Cisco Systems. [Online]. Available: https://www.netacad.com/about-networking-academy/packet-tracer/. Accessed May. 28, 2016.

[25] Oracle VM Virtualbox. (n.d.). Oracle. [Online]. Available: https://www.virtualbox.org/wiki/VirtualBox. Accessed Jun. 6, 2016.

[26] Quagga Routing Suite. (n.d.). quagga. [Online]. Available: http://www.nongnu.org/quagga/. Accessed May. 12, 2016.

[27] GNS3. (n.d.). GNS3. [Online]. Available: https://www.gns3.com/. Accessed Apr. 6, 2016.

[28] E. Rye, "Evaluating the limits of network topology inference via virtualized network emulation," M.S. thesis, Computer Science, Naval Postgraduate School, Monterey, California, 2015.

[29] H. Nguyen, M. Roughan, S. Knight, N. Falkner, O. Maennel, and R. Bush, "How to build complex, large-scale emulated networks," in *Testbeds and Research Infrastructures. Development of Networks and Communities*. Springer, 2010, pp. 3–18.

[30] S. Knight, A. Jaboldinov, O. Maennel, I. Phillips, and M. Roughan, "Autonetkit: simplifying large scale, open-source network experimentation," in *Proceedings of the ACM SIGCOMM 2012 conference on Applications, technologies, architectures, and protocols for computer communication*. ACM, 2012, pp. 97–98.

[31] M. Luckie, "Scamper: A scalable and extensible packet prober for active measurement of the internet," in *Proceedings of the 10th ACM SIGCOMM conference on Internet measurement*. ACM, 2010, pp. 239–245.

[32] S. Guppy. (2015, Jan. 18). GNS3 Community News and Discussions. [Online]. Available: https://gns3.com/discussions/gns3-in-two-computers/

[33] K. Yocum, E. Eade, J. Degesys, D. Becker, J. Chase, and A. Vahdat, "Toward scaling network emulation using topology partitioning," in *Modeling, Analysis and Simulation of Computer Telecommunications Systems, 2003. MASCOTS 2003. 11th IEEE/ACM International Symposium on*. IEEE, 2003, pp. 242–245.

[34] P. Ciarlet Jr and F. Lamour, "On the validity of a front-oriented approach to partitioning large sparse graphs with a connectivity constraint," *Numerical Algorithms*, vol. 12, no. 1, pp. 193–214, 1996.

[35] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on scientific Computing*, vol. 20, no. 1, pp. 359–392, 1998.

[36] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker, "Scalability and accuracy in a large-scale network emulator," *ACM SIGOPS Operating Systems Review*, vol. 36, no. SI, pp. 271–284, 2002.

[37] R. L. Rardin, *Optimization in operations research*. Prentice Hall: Upper Saddle River, NJ, 1998, vol. 166.

[38] COmputational INfrastructure for Operations Research. (n.d.). COmputational INfrastructure for Operations Research. [Online]. Available: http://www.coin-or.org/. Accessed Apr. 12, 2016.

[39] GLPK: GNU Project: Free Software Foundation. (n.d.). The Free Software Foundation. [Online]. Available: https://www.gnu.org/software/glpk/. Accessed May. 29, 2016.

[40] Gurobi Optimization: The Best Mathamatical Programming Solver. (n.d.). Gurobi. [Online]. Available: http://www.gurobi.com/index. Access May. 29, 2016.

[41] G. Brown and R. Dell, "Formulating linear and integer linear programs: A rouges' gallery," *INFORMS Trans. Ed*, vol. 7, no. 2, 2007.

[42] The CAIDA AS Relationships Dataset, 20160501. (n.d.). Center for Applied Internet Data Analysis. [Online]. Available: http://www.caida.org/data/active/as-relationships/. Accessed May. 31, 2016.

[43] bridge. (n.d.). The Linux Foundation. [Online]. Available: http://www.linuxfoundation.org/collaborate/workgroups/networking/bridge. Accessed May. 20, 2016.

[44] M. Krasnyansky. (n.d.). Universal TUN/TAP device driver. [Online]. Available: https://www.kernel.org/doc/Documentation/networking/tuntap.txt. Accessed May. 16, 2016.

[45] D. Libes. (n.d.). The Expect Home Page. [Online]. Available: http://expect.sourceforge.net/. Accessed Apr. 15, 2016.

[46] Tinc VPN. (n.d.). tinc. [Online]. Available: https://www.tinc-vpn.org/. Accessed: Apr. 18, 2016.

[47] J. Forcier. (n.d.). Fabric: Pythonic remote exeuction. [Online]. Available: http://www.fabfile.org/. Accessed Apr. 14, 2016.

[48] Jinja2: The Python Template Engine. (n.d.). Jinja. [Online]. Available: http://jinja.pocoo.org/. Accessed Mar. 28, 2016.

[49] PuLP v1.4.6 documentation. (n.d.). Computational Infrastructure for Operations Research. [Online]. Available: http://www.coin-or.org/PuLP/main/installing_pulp_at_home.html. Accessed Apr. 12, 2016.

[50] IPv4:IPv6 manipulation library. (n.d.). The Python Software Foundation. [Online]. Available: https://docs.python.org/3/library/ipaddress.html. Accessed Mar. 29, 2016.

[51] A. A. Hagberg, D. A. Schult, and P. J. Swart, "Exploring Network Structure, Dynamics, and Function using NetworkX," in *Proceedings of the 7th Python in Science Conference (SciPy2008)*, Pasadena, CA, Aug. 2008, pp. 11–15.

[52] Dynamips / Dynagen Tutorial. (n.d.). [Online]. Available: http://www.iteasypass.com/Dynamips.htm. Accessed Apr. 16, 2016.

[53] T. Y. James, "Performance evaluation of linux bridge," in *Telecommunications System Management Conference*, 2004.

[54] E. Freeman, E. Robson, B. Bates, and K. Sierra, *Head first design patterns*. O'Reilly Media, Inc., 2004.

[55] D. S. Johnson, "Near-optimal bin packing algorithms," Ph.D. dissertation, Dept. Mathematics, Massachusetts Institute of Technology, Cambridge, MA, 1973.

[56] M. Newman, *Networks: An Introduction*. Oxford, United Kingdom: Oxford University Press, 2010.

[57] TimeComplexity: Python Wiki. (n.d.). Python Software Foundation. [Online]. Available: https://wiki.python.org/moin/TimeComplexity. Accessed May. 19, 2016.

[58] X. Shi, Y. Xiang, Z. Wang, X. Yin, and J. Wu, "Detecting prefix hijackings in the internet with argus," in *Proceedings of the 2012 ACM conference on Internet measurement conference*. ACM, 2012, pp. 15–28.

[59] BIRD Internet routing daemon. (2013). [Online]. Available: http://bird.network.cz/

[60] pybgpdump. (2007). [Online]. Available: https://jon.oberheide.org/pybgpdump/

[61] D. Song. (n.d.). dpkt. [Online]. Available: https://github.com/kbandla/dpkt. Accessed Jun. 8, 2016.

[62] CPLEX Optimizer. (n.d.). IBM. [Online]. Available: http://www-01.ibm.com/software/commerce/optimization/cplex-optimizer/. Accessed May. 24, 2016.

[63] C. Orsini, A. King, and A. Dainotti, "Bgpstream: A software framework for live and historical bgp data analysis."

THIS PAGE INTENTIONALLY LEFT BLANK

Initial Distribution List

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California